

Peter Norton's **Machinetaalboek** **voor de IBM-PC**



Peter Norton
John Socha

Kluwer

**Peter Norton's
Machinetaalboek
voor de
IBM-PC**

MICHELIN 11/90
BIBLIOTHEEK
VOLGNR. 820 1

Peter Norton / John Socha

**Peter Norton's
Machinetaalboek
voor de
IBM-PC**



Kluwer Technische Boeken B.V. / Deventer-Antwerpen

vertaling: Jan Kuis

ISBN 90 201 2165 0

D/1988/0108/153

NUGI 856

Oorspronkelijke titel: Peter Norton's Assembly Language Book for the IBM PC
Uitgegeven bij: Brady Books, Verenigde Staten

© 1986 by Brady Communications Company, Inc.

© 1989 van de Nederlandse vertaling bij Kluwer Technische Boeken B.V.-Deventer

1^e druk 1989

2^e oplage 1990

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand, of openbaar gemaakt, in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen, of enige andere manier, zonder voorafgaande schriftelijke toestemming van de uitgever.

Voor zover het maken van kopieën uit deze uitgave is toegestaan op grond van artikel 16B Auteurswet 1912 j° het Besluit van 20 juni 1974, Stb. 351, zoals gewijzigd bij het Besluit van 23 augustus 1985, Stb. 471 en artikel 17 Auteurswet 1912, dient men de daarvoor wettelijk verschuldigde vergoedingen te voldoen aan de Stichting Reprorecht (Postbus 882, 1180 AW Amstelveen). Voor het overnemen van gedeelte(n) uit deze uitgave in bloemlezingen, readers en andere compilatiewerken (artikel 16 Auteurswet 1912) dient men zich tot de uitgever te wenden.

All rights reserved. No part of this book may be reproduced, stored in a database or retrieval system, or published, in any form or in any way, electronically, mechanically, by print, photoprint, microfilm or any other means without prior written permission from the publisher.

Ondanks alle aan de samenstelling van de tekst bestede zorg, kan noch de redactie noch de uitgever aansprakelijkheid aanvaarden voor eventuele schade, die zou kunnen voortvloeien uit enige fout, die in deze uitgave zou kunnen voorkomen.

Inhoud

Deel 1 Machinetaal

- 1 Debug en rekenen 17**
 - 1.1 Hexadecimale getallen 18
 - 1.2 Debug 18
 - 1.3 Hexarithmetiek 19
 - 1.4 Hexadecimaal omzetten in decimaal 21
 - 1.5 Hex-getallen van vijf cijfers 24
 - 1.6 Decimaal omzetten in hex 25
 - 1.7 Negatieve getallen 27
 - 1.8 Bits, bytes, woorden en binaire notatie 28
 - 1.9 Twee-complement — een raar soort negatief getal 30
 - 1.10 Samenvatting 32
- 2 Rekenen en de 8088 33**
 - 2.1 Registers als variabelen 34
 - 2.2 Geheugen en de 8088 35
 - 2.3 Optellen volgens de 8088 37
 - 2.4 Aftrekken volgens de 8088 39
 - 2.5 Negatieve getallen in de 8088 40
 - 2.6 Bytes in de 8088 40
 - 2.7 Vermenigvuldigen en delen volgens de 8088 41
 - 2.8 Samenvatting 44
- 3 Tekens afbeelden 46**
 - 3.1 INT — de krachtige interrupt 47
 - 3.2 Een net einde — INT 20h 49
 - 3.3 Een programma van twee regels — de stukjes bij elkaar leggen 49
 - 3.4 Programma's invoeren 50
 - 3.5 Gegevens in registers zetten 51
 - 3.6 Een reeks tekens schrijven 53
 - 3.7 Samenvatting 55
- 4 Binaire getallen afdrukken 57**
 - 4.1 Rotaties en de overdrachtvlag 58
 - 4.2 Optellen met de overdrachtvlag 59
 - 4.3 Lussen 60
 - 4.4 Een binair getal schrijven 62
 - 4.5 De *Proceed*-opdracht 63
 - 4.6 Samenvatting 64

5	Afdrukken in hex	65
5.1	Vergelijk- en statusbits	66
5.2	Een enkel hex-cijfer afdrukken	68
5.3	Nog een roteer-instructie	71
5.4	Logica en AND	72
5.5	Alles bij elkaar leggen	73
5.6	Samenvatting	74
6	Tekens lezen	75
6.1	Een teken lezen	76
6.2	Een hex-getal van een cijfer lezen	76
6.3	Een hex-getal van twee cijfers lezen	77
6.4	Samenvatting	78
7	Procedures — verwant met subroutines	79
7.1	Procedures	80
7.2	De stapel en terugkeeradressen	81
7.3	PUSH en POP	83
7.4	Gemakkelijker hex-getallen lezen	85
7.5	Samenvatting	87

Deel 2 Assembleertaal

8	Welkom bij de assembler	91
8.1	Een programma zonder Debug	92
8.2	Aanmaken van bronbestanden	95
8.3	Linken	95
8.4	Terug naar Debug	97
8.5	Commentaar	97
8.6	Labels	98
8.7	Samenvatting	100
9	Procedures en de assembler	101
9.1	De procedures van de assembler	102
9.2	De procedures voor hex-uitvoer	104
9.3	De beginselen van modulair ontwerpen	108
9.4	Raamwerk van een programma	108
9.5	Samenvatting	109
10	Afdrukken in decimaal	111
10.1	De conversie	112
10.2	Enkele trucs	114
10.3	Interne zaken	116
10.4	Samenvatting	117

- 11 Segmenten 119**
 - 11.1 Het geheugen van de 8088 in stukken verdelen 120
 - 11.2 Pseudo-ops voor segmenten 126
 - 11.3 De pseudo-op ASSUME 127
 - 11.4 NEAR- en FAR-aanroepen 128
 - 11.5 Meer over de INT-instructie 130
 - 11.6 Interrupt-vectoren 131
 - 11.7 Samenvatting 132
- 12 Koerswijzigingen 133**
 - 12.1 Schijven, sectoren en Dskpatch 134
 - 12.2 De aanpak 136
 - 12.3 Samenvatting 137
- 13 Modulair ontwerpen 139**
 - 13.1 Afzonderlijk assembleren 140
 - 13.2 De drie wetten van het modulair ontwerpen 143
 - 13.3 Samenvatting 146
- 14 Geheugen dumpen 147**
 - 14.1 Adresseermodi 148
 - 14.2 Tekens aan de dump toevoegen 153
 - 14.3 256 bytes geheugen dumpen 155
 - 14.4 Samenvatting 159
- 15 Een schijfsector dumpen 161**
 - 15.1 Het leven gemakkelijker maken 162
 - 15.2 Indeling van het Make-bestand 162
 - 15.3 Toon sec opknappen 163
 - 15.4 Een sector lezen 165
 - 15.5 Samenvatting 169
- 16 Verfraaien van de sector-afbeelding 171**
 - 16.1 Grafische tekens toevoegen 172
 - 16.2 Adressen aan de afbeelding toevoegen 174
 - 16.3 Horizontale lijnen toevoegen 177
 - 16.4 Getallen aan de afbeelding toevoegen 181
 - 16.5 Samenvatting 183

Deel 3 Het ROM BIOS van de IBM PC

- 17 De ROM BIOS-routines 187**
 - 17.1 VIDEO_IO, de ROM BIOS-routines 188
 - 17.2 Verplaatsen van de cursor 193
 - 17.3 Ander gebruik van variabelen 194
 - 17.4 De kopregel schrijven 198
 - 17.5 Samenvatting 200

18	De definitieve SCHRIJF__TEK	203
18.1	Een nieuwe SCHRIJF__TEK	204
18.2	Wissen tot het einde van een regel	207
18.3	Samenvatting	209
19	De verdeelprocedure	211
19.1	De Verdeler	211
19.2	Andere sectoren lezen	217
19.3	Opzet van de volgende hoofdstukken	219
20	Een programmeer-uitdaging	221
20.1	De fantoomcursors	222
20.2	Eenvoudige wijzigingen	223
20.3	Andere toevoegingen en wijzigingen in Dskpatch	224
21	De fantoomcursors	225
21.1	De fantoomcursors	226
21.2	Tekenattributen veranderen	231
21.3	Samenvatting	232
22	Eenvoudige wijzigingen	233
22.1	De fantoomcursors verplaatsen	234
22.2	Eenvoudige wijzigingen	236
22.3	Samenvatting	240
23	Invoer hex en decimaal	241
23.1	Hex-invoer	242
23.2	Decimale invoer	248
23.3	Samenvatting	251
24	Verbeterde toetsenbord-invoer	253
24.1	Een nieuwe LEES__STRING	254
25	Op zoek naar fouten	261
25.1	Verbeteren van VERDELER	262
25.2	Samenvatting	263
26	Gewijzigde sectoren wegschrijven	265
26.1	Naar de schijf schrijven	266
26.2	Andere methoden van debuggen	268
26.3	Een laadoverzicht opzetten	268
26.4	Fouten opsporen	270
26.5	Symdeb	272
26.5.1	Symbolisch debuggen	272
26.5.2	Schermomwisseling	272
26.6	Samenvatting	274

- 27 De andere halve sector 277**
- 27.1 Een halve sector verschuiven 278
- 27.2 Samenvatting 280

Deel 4 Losse eindjes

- 28 Verplaatsen 283**
- 28.1 Meervoudige segmenten 284
- 28.2 Verplaatsing 287
- 28.3 .COM- contra .EXE-programma's 290
- 29 Meer over segmenten en ASSUME 293**
- 29.1 Segment override 294
- 29.2 Nog een blik op ASSUME 296
- 29.3 Phase errors 296
- 29.4 Slotopmerkingen 297

Appendix A Handleiding bij de diskette 299

- A.1 Voorbeelden uit de hoofdstukken 300
- A.2 Uitgebreide versie van Dskpatch 300

Appendix B Listing van Dskpatch 305

- B.1 Beschrijvingen van procedures 306
- B.2 Programmalistings van Dskpatch-procedures 313
- B.2.1 Make-bestand DSKPATCH 313
- B.2.2 CURSOR.ASM 314
- B.2.3 DISK_IO.ASM 317
- B.2.4 DSKPATCH.ASM 320
- B.2.5 EDITOR.ASM 322
- B.2.6 FANTOOM.ASM 324
- B.2.7 TBD IO.ASM 330
- B.2.8 TOON SEC.ASM 337
- B.2.9 VERDEEL.ASM 341
- B.2.10 VIDEO_IO.ASM 343

Appendix C Laadvolgorde segmenten 349

- C.1 Laadvolgorde segmenten 350
- C.2 Phase errors 352
- C.3 EXE2BIN File cannot be converted 354

Appendix D Veel voorkomende foutmeldingen 355

- D.1 MASM 356
- D.2 LINK 357
- D.3 EXE2BIN 358

Appendix E Tabellen 359

- E-1. ASCII-tekencodes 360
- E-2. Uitgebreide toetsenbordcodes 362
- E-3. De adresseermodi 363
- E-4. Functies van INT 10h 364
- E-5. Functies van INT 21h 367

Trefwoordenregister 369

Inleiding

Wanneer u dit boek hebt doorgewerkt, weet u hoe u volledige programma's in assembleertaal kunt schrijven: tekstbewerkers, utility's enz. U zult daarbij allerlei technieken leren die beroepsprogrammeurs gebruiken om hun werk te vereenvoudigen. Deze technieken, waaronder modulaire opbouw en stapsgewijs verfijnen, zullen uw programmeersnelheid verdubbelen of verdrievoudigen, en u helpen snellere en betrouwbaardere programma's te schrijven.

Vooraf de techniek van het stapsgewijs verfijnen scheelt veel werk bij het schrijven van ingewikkelde programma's. Als u ooit dat ontmoedigende gevoel hebt gehad van 'waar moet ik beginnen?', zult u merken dat stapsgewijs verfijnen een eenvoudige en natuurlijke manier is om programma's te schrijven. En het is nog leuk ook.

Dit boek bestaat echter niet alleen uit theorie. We schrijven ook een programma. Het heet *Dskpatch* en u zult het om diverse redenen een nuttig programma vinden. Allereerst zult u zien hoe stapsgewijze verfijning en modulaire opbouw voor een echt programma worden toegepast, zodat u de kans krijgt om te zien waarom die technieken zo nuttig zijn. Bovendien is *Dskpatch*, zoals u al gauw zult merken, op zichzelf ook een algemeen, volledig scherm-programma waarmee u schijfsectoren kunt wijzigen en dat u ook lang nadat u het boek uit hebt nog in z'n geheel zowel als deels kunt gebruiken.

Waarom assembleertaal?

We nemen aan dat u dit boek hebt gekozen omdat u geïnteresseerd bent in assembleertaal. Maar u weet misschien niet helemaal zeker waarom u het zou willen leren. Een reden die misschien het minst voor de hand ligt, is dat assembleerprogramma's de kern van elke IBM-PC of compatible vormen. Assembleertaal is de grootste gemeene deler van alle andere programmeertalen. Het brengt je dicht bij de machine dan hogere programmeertalen, dus als u assembleertaal leert, krijgt u ook meteen inzicht in de 8088-microprocessor binnenin uw computer. We zullen u net als de schrijvers van andere leerboeken de instructies van de 8088-microprocessor leren, maar we gaan veel verder en bespreken ook leerstof voor gevorderden die van onschatbare waarde zal blijken te zijn wanneer u uw eigen programma's begint te schrijven.

Hebt u eenmaal inzicht in de 8088-microprocessor in uw eigen IBM-PC, dan zult u veel van wat u in andere programma's en hoge programmeertalen ziet beter gaan begrijpen. Misschien is het u bijvoorbeeld opgevallen dat de grootste integer die u in BASIC kunt hebben, 32767 is. Waar komt dat getal vandaan? Het is een vreemd getal voor een bovengrens. Maar zoals u verderop zult zien, houdt het getal 32767 rechtstreeks verband met de wijze waarop uw IBM-PC getallen opslaat.

Misschien bent u ook geïnteresseerd in snelheid of grootte. In de regel zijn assembleerprogramma's veel sneller dan in elke andere taal. Assembleerprogramma's zijn gemiddeld twee- tot driemaal zo snel als overeenkomstige programma's in C of Pascal, en vijftien keer zo snel als geïnterpreteerde BASIC-programma's. Assembleerprogramma's zijn meestal ook kleiner. Het *Dskpatch*-programma dat we in dit boek gaan opbouwen, zal met ongeveer één kilobyte helemaal operationeel zijn. Vergelijken met programma's in het algemeen, is dat klein. Een vergelijkbaar programma in C of Pascal zou ongeveer tien keer zo lang zijn. Om die reden, en andere, heeft

het bedrijf Lotus zijn 1-2-3-programma geheel in assembleertaal geschreven. Programma's in assembleertaal bieden u ook volledig toegang tot alle mogelijkheden van uw computer. Een aantal programma's, zoals Sidekick, ProKey en SuperKey, blijven in het geheugen als u ze hebt gestart. Dergelijke programma's veranderen de manier waarop uw machine werkt, en gebruiken systeemeigenschappen waarover alleen assembleerprogramma's kunnen beschikken.

Dskpatch

In ons werk met assembleertaal zullen we rechtstreeks schijfsectoren bekijken en tekens en getallen afbeelden (in hexadecimale notatie) die DOS daar heeft opgeslagen. Dskpatch is een volledige scherm-editor waarmee deze tekens en getallen in een schijfsector kunnen worden bewerkt. Met Dskpatch kunt u bijvoorbeeld de sector bekijken waarop DOS de directory van een schijf opslaat en kunt u bestandsnamen of andere informatie wijzigen. Op die manier krijgt u een goed inzicht in hoe DOS informatie op een schijf opslaat.

Dskpatch is echter méér dan gewoon een programma. Dskpatch bevat zo'n 50 subroutines. U zult merken dat veel van die algemene subroutines nuttig zijn wanneer u uw eigen programma's schrijft. Op die manier biedt dit boek niet alleen een inleiding tot het gebruik van de 8088 en het programmeren in assembleertaal, maar is het ook een bron van nuttige subroutines.

Bovendien moet elke volledige scherm-editor gebruik maken van de specifieke mogelijkheden van de IBM-PC-computerfamilie. Met de voorbeelden in dit boek leert u ook nuttige programma's schrijven voor IBM-PC's, AT's of daarmee compatibele computers.

Vereiste apparatuur

Welke apparatuur hebt u nodig om de voorbeelden in dit boek te kunnen draaien? Allereerst een IBM-PC of daarmee compatibele computer met ten minste 128K geheugen en een diskdrive. Ook moet u de PC-DOS (of MS-DOS) versie 2.00 of hoger hebben. En vanaf deel 2 hebt u de IBM of de Microsoft Macro Assembler nodig.

Opzet van dit boek

Dit boek is onderverdeeld in vier delen, elk met de nadruk op verschillende aspecten. Of u al of niet iets van microprocessoren weet, en of u al of niet een assembleertaal beheerst, u zult in elk geval stukken vinden die voor u van belang zijn.

Deel 1 richt zich op de 8088-microprocessor. Hierin vindt u de geheimen van bits, bytes en machinetaal. Elk van de zeven hoofdstukken bevat een overvloed aan echte voorbeelden waarbij gebruik wordt gemaakt van een programma met de naam Debug, dat op uw DOS-schijf staat. Met Debug kunt u *in* die befaamde 8088-microprocessor kijken, die in uw PC zit terwijl hij DOS draait. Bij deel 1 wordt ervan uitgegaan dat u een elementaire kennis van BASIC bezit en met uw computer overweg kunt.

Deel 2, de hoofdstukken 8 tot en met 16, gaat over assembleertaal en hoe u programma's met de assembler kunt schrijven. Het wordt heel geleidelijk aangepakt, en in

plaats van alle onderdelen van de assembler zelf te bespreken, richten we ons op een aantal assembleeropdrachten die we nodig hebben om nuttige programma's te schrijven.

We zullen de assembler gebruiken om enkele programma's van deel 1 te herschrijven, en beginnen dan met het schrijven van Dskpatch. We bouwen dit programma langzaam op, zodat u de stapsgewijze verfijning bij het opzetten van grote programma's leert toepassen. Ook bespreken we technieken als modulair ontwerpen die u helpen bij het schrijven van duidelijke programma's. Zoals gezegd maken zulke technieken het programmeren eenvoudiger omdat ze de ingewikkelde toestanden die gewoonlijk bij het schrijven van assembleerprogramma's horen, helpen vermijden.

In deel 3, met de hoofdstukken 17 tot en met 27, richten we ons op de toepassing van meer ingewikkelde mogelijkheden van IBM-PC's, zoals het verplaatsen van de cursor en het schoonmaken van het scherm.

In deel 3 bespreken we ook methoden om fouten in grotere assembleerprogramma's op te sporen. Assembleerprogramma's groeien erg snel en zijn al gauw twee of meer pagina's lang zonder dat ze erg veel doen (Dskpatch wordt langer). Al passen we die foutopsporingmethoden toe op programma's die groter zijn dan enkele pagina's, ze zijn ook handig bij kleine programma's.

Ten slotte worden in deel 4 nog enkele losse eindjes aan elkaar geknoopt.

Laten we ons nu zonder verdere omhaal op de 8088 werpen en bekijken hoe hij getallen opslaat.

DEEL 1

Machinetaal

1 Debug en rekenen

- 1.1 Hexadecimale getallen 18**
- 1.2 Debug 18**
- 1.3 Hexarithmetic 19**
- 1.4 Hexadecimaal omzetten in decimaal 21**
- 1.5 Hex-getallen van vijf cijfers 24**
- 1.6 Decimaal omzetten in hex 25**
- 1.7 Negatieve getallen 27**
- 1.8 Bits, bytes, woorden en binaire notatie 28**
- 1.9 Twee-complement — een raar soort negatief
getal 30**
- 1.10 Samenvatting 32**

Laten we onze ontdekkingsreis in de assembleertaal beginnen met te leren hoe computers tellen. Dat klinkt misschien heel eenvoudig. Per slot tellen we tot 11 door met 1 te beginnen en dan steeds door te tellen: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

Maar een computer telt niet zo. Tot vijf tellen doet hij zo: 1, 10, 11, 100, 101. De getallen 10, 11, 100 enzovoort zijn binaire getallen, gebaseerd op een talstelsel met maar twee cijfers, 0 en 1, in plaats van de tien zoals we die kennen van ons vertrouwde tientallige stelsel. Het binaire getal 10 is op deze manier gelijk aan het decimale getal dat wij kennen als 2.

Binaire getallen zijn voor ons belangrijk omdat ze de vorm zijn waarin de 8088-microprocessor getallen binnenin uw IBM-PC gebruikt. Maar hoewel computers het van die binaire getallen moeten hebben, kan het vervelend lang duren om al die reeksen enen en nullen uit te schrijven. De oplossing? Hexadecimale getallen — een veel compactere manier om binaire getallen te schrijven. In dit hoofdstuk leert u beide manieren om getallen te schrijven: hexadecimaal en binair. En terwijl u leert hoe computers tellen, leert u ook hoe ze getallen opslaan — in bits, bytes en woorden. Als u al bekend bent met binaire en hexadecimale getallen, bits, bytes en woorden, kunt u nu meteen doorgaan naar de samenvatting van dit hoofdstuk.

1.1 Hexadecimale getallen

Omdat hexadecimale getallen gemakkelijker in het gebruik zijn dan binaire getallen — althans wat hun lengte betreft — beginnen we met de hexadecimale (afgekort hex), en gebruiken we DEBUG.COM, een programma dat u op uw supplementaire DOS-schijf zult aantreffen. We zullen Debug hier en in volgende hoofdstukken gebruiken om machinetaalprogramma's met één instructie tegelijk in te voeren en te draaien. Net als BASIC werkt Debug in een prettige, interactieve omgeving. Maar anders dan BASIC kent het geen decimale getallen. Voor Debug is het getal 10 een hexadecimaal getal — niet tien. En omdat Debug alleen hexadecimaal spreekt, zult u iets over hex-getallen moeten weten. Maar laten we eerst even een zijspoor inslaan en iets over Debug zelf te weten komen.

1.2 Debug

Waarom heeft dit programma de naam Debug? Een *bug* is in de computerwereld een fout in een programma. Een programma dat goed werkt, heeft geen *bugs*, terwijl een programma dat hapert minstens één fout bevat. Door Debug te gebruiken om een programma met één instructie tegelijk te draaien en te kijken hoe het programma werkt, kunnen we fouten erin opsporen en herstellen. Dit staat bekend als *debuggen*, van fouten ontdoen — vandaar de naam Debug.

Volgens de computerverhalen is het woord debuggen ontstaan in de begintijd van de computers — met name op een dag dat de Mark I op Harvard niet werkte. Na lang zoeken vonden de technici toen de oorzaak van hun moeilijkheden: een insect dat vastzat tussen de contacten van een relais. De technici haalden het insect weg en schreven in het werkverslag over het weghalen van een insect (*bug*) in de Mark I. Als u met twee drives werkt, zoek dan Debug op uw supplementaire DOS-schijf op en we kunnen beginnen. U moet ook een werkschijf bij de hand hebben, en daar DEBUG.COM op kopiëren. Als u een harde schijf hebt, staat Debug misschien al

in de DOS-directory of kunt u hem daarin zetten. We zullen Debug veel gebruiken in deel 1 van dit boek.

N.B. Vanaf nu staat de tekst die u in interactieve sessies als deze tikt op een gekleurde achtergrond om hem te onderscheiden van de respons van uw computer. Tik de tekst, druk de Enter-toets in en u moet eenzelfde reactie krijgen als die we in de sessies laten zien. U zult niet altijd precies dezelfde respons krijgen omdat uw computer vermoedelijk een andere hoeveelheid geheugen heeft dan de computer waarop wij dit boek hebben geschreven. (In het volgende hoofdstuk zult u al een paar van die verschillen tegenkomen.) Merk bovendien op dat we in alle voorbeelden hoofdletters gebruiken. Dit is alleen gedaan om verwarring van de kleine letter l (el) met het cijfer 1 (een) te voorkomen. Als u dat liever doet, mag u gerust alle voorbeelden met kleine letters tikken.

Start na deze afspraken Debug door na de DOS-prompt (die in dit voorbeeld de vorm C> heeft) de naam te tikken:

```
C>DEBUG
```

Het koppelstreepje dat u als antwoord op uw opdracht ziet, is de prompt van Debug, net zoals C> een DOS-prompt is. Het betekent dat Debug wacht op een opdracht. Als u met Debug wilt stoppen en terugkeren naar DOS, tikt u gewoon Q (van *Quit*) achter het streepje en drukt op Enter. Probeer, als u wilt, nu maar te stoppen en keer dan terug naar Debug:

```
-Q  
C>DEBUG
```

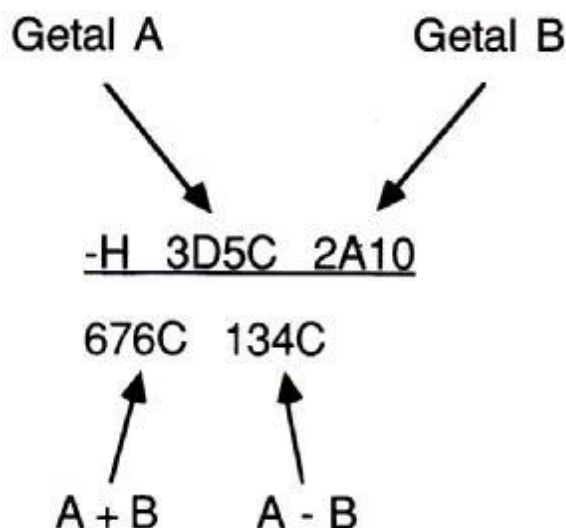
Nu kunnen we verder met het leren over hex-getallen.

1.3 Hexarithmetic

We gaan een Debug-opdracht genaamd H gebruiken. H is een afkorting van *Hexarithmetic* (rekenen in hex) en wordt gebruikt om twee hex-getallen op te tellen of van elkaar af te trekken. Laten we eens zien hoe H werkt als we 2 en 3 willen optellen. We weten dat bij decimale getallen $2 + 3$ gelijk aan 5 is. Geldt dat ook bij hex-getallen? Kijk of u nog in Debug zit, en tik na het prompt-streepje het volgende:

```
-H 3 2  
0005 0001
```

Debug drukt de som (0005) en het verschil (0001) van 3 en 2 af. De H-opdracht berekent altijd de som en het verschil van twee getallen, zoals hier. En tot dusver is de uitkomst bij hex- en decimale getallen hetzelfde: 5 is de som van 3 en 2 in decimaal, en 1 is het verschil ($3 - 2$). Maar soms kunt u voor verrassingen komen te staan.



Afb. 1-1. De H-opdracht.

Stel dat we nu *H 2 3* hadden getikt om de twee getallen op te tellen en af te trekken. Als we dat proberen:

```
-H 2 3
0005 FFFF
```

krijgen we FFFF in plaats van -1, voor 2 - 3. Vreemd als het mag lijken is FFFF toch een getal: het is hex voor -1.

We komen zo nog terug op deze ongebruikelijke -1. Maar eerst kijken we even naar iets grotere getallen om te zien hoe een F in een getal kan voorkomen.

Om te zien hoe de H-opdracht werkt bij grotere getallen, proberen we negen plus een, wat als uitkomst het getal 10 moet geven:

```
-H 9 1
000A 0008
```

Negen plus een is A? Klopt: A is het hex-getal voor tien. En wat gebeurt er nu als we een nog groter getal proberen, zoals 15:

```
-H 9 6
000F 0003
```

Als u andere getallen tussen tien en vijftien probeert, zult u alles bij elkaar 16 cijfers zien — 0 tot en met F (0 tot en met 9 en A tot en met F). De naam hexadecimaal komt van hexa- (zes) en deca- (tien), wat samen 16 voorstelt. De cijfers 0 tot en met 9 zijn in hexadecimaal en decimaal hetzelfde; de hexadecimale cijfers A tot en met F zijn gelijk aan de decimale getallen 10 tot en met 15.

Waarom spreekt Debug hexadecimaal? U zult zo zien dat we met twee hex-cijfers 256 verschillende getallen kunnen schrijven. Zoals u misschien al vermoedde, heeft 256 ook iets te maken met de eenheid die we kennen als een byte, en de byte speelt in

computers en in dit boek een belangrijke rol. Aan het eind van dit hoofdstuk komt u meer te weten over bytes, maar voorlopig gaan we weer verder met de hex-getallen, de enige getallen die Debug kent, en het rekenen in hex.

1.4 Hexadecimaal omzetten in decimaal

Tot dusver hebben we alleen gekeken naar hex-getallen van één cijfer. We gaan nu zien hoe grotere hex-getallen worden voorgesteld, en hoe die getallen in decimale getallen worden omgezet.

Net als bij decimale getallen, bouwen we hex-getallen van meerdere cijfers op door er aan de linkerkant meer cijfers bij te zetten. Stel dat we het getal 1 willen optellen bij het grootste decimale getal van één cijfer: 9. De uitkomst is dan een getal van twee cijfers, 10 (tien). Wat gebeurt er nu wanneer we 1 optellen bij het grootste hex-getal van één cijfer, F? Dan krijgen we weer tien.

<u>Decimaal</u>	<u>Hex-cijfer</u>
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Afb. 1-2. Hexadecimale cijfers.

Maar wacht, tien in hex is eigenlijk 16, niet tien. Dat kan een hoop verwarring geven. We moeten een manier hebben om die twee tienen uit elkaar te houden; daarom zetten we van nu af achter elk hex-getal de letter h. Zodoende weten we dat 10h hexadecimaal 16 is, en 10 decimaal tien.

Nu komen we bij de vraag hoe we getallen van hex naar decimaal kunnen omzetten. We weten dat 10h gelijk is aan 16, maar hoe zetten we een groter hex-getal, zoals

D3h, om in een decimaal getal zonder van 10h door te tellen naar D3h? Of, hoe zetten we het decimale getal 173 om in hex?

We kunnen Debug niet gebruiken als hulpmiddel, want Debug spreekt geen decimaal. In hoofdstuk 10 zullen we een programma schrijven waarmee een hex-getal in de decimale notatie wordt omgezet zodat onze programma's in decimaal met ons kunnen praten. Maar op dit moment zullen we de omzettingen met de hand moeten doen, dus laten we eerst terugkeren naar ons vertrouwde wereldje van decimale getallen.

Wat betekent het getal 276 eigenlijk? Op de lagere school hebben we geleerd dat 276 betekent dat we twee keer honderd hebben, zeven keer tien en zes keer een. Of, afgebeeld:

$$\begin{array}{r} 2 \quad * \quad 100 = 200 \\ 7 \quad * \quad 10 = 70 \\ 6 \quad * \quad 1 = 6 \\ \hline 276 = 276 \end{array}$$

Dat laat in elk geval zien wat die cijfers betekenen. Kunnen we dezelfde figuur gebruiken voor een hex-getal? Uiteraard.

Bekijk het getal D3h eens, waar we het net over hadden. D is het hexadecimale cijfer 13, en er zijn 16 hex-cijfers, tegen 10 decimale, dus D3h is dertien maal zestien plus drie keer een. Dat kan zo worden afgebeeld:

$$\begin{array}{r} D \rightarrow 13 * 16 = 208 \\ 3 \rightarrow 3 * 1 = 3 \\ \hline D3h = 211 \end{array}$$

Bij het decimale getal 276 vermenigvuldigden we de cijfers met 100, 10 en 1; bij het hex-getal D3 vermenigvuldigden we de cijfers met 16 en 1. Als we vier decimale cijfers hadden gehad, zouden we met 1000, 100, 10 en 1 hebben vermenigvuldigd. Welke vier getallen zouden we bij vier hex-cijfers gebruiken?

Bij decimale getallen zijn de getallen 1000, 100, 10 en 1 allemaal machten van 10:

$$\begin{array}{l} 10^3 = 1000 \\ 10^2 = 100 \\ 10^1 = 10 \\ 10^0 = 1 \end{array}$$

Bij hex-cijfers kunnen we precies dezelfde methode gebruiken, maar dan met machten van 16 in plaats van 10, dus onze vier getallen worden dan:

$$\begin{array}{l} 16^3 = 4096 \\ 16^2 = 256 \\ 16^1 = 16 \\ 16^0 = 1 \end{array}$$

Laten we nu 3AC8h eens omzetten naar decimaal met behulp van de vier getallen die we net hebben berekend:

$$\begin{array}{rcl}
 3 & \rightarrow & 3 * 4096 = 12288 \\
 A & \rightarrow & 10 * 256 = 2560 \\
 C & \rightarrow & 12 * 16 = 192 \\
 8 & \rightarrow & 8 * 1 = 8 \\
 \hline
 3AC8h & & = 15048
 \end{array}$$

$$\begin{array}{rcl}
 7 & \rightarrow & 7 * 16 = 112 \\
 C & \rightarrow & 12 * 1 = 12 \\
 \hline
 7Ch & & = 124
 \end{array}$$

$$\begin{array}{rcl}
 3 & \rightarrow & 3 * 256 = 768 \\
 F & \rightarrow & 15 * 16 = 240 \\
 9 & \rightarrow & 9 * 1 = 9 \\
 \hline
 3F9h & & = 1,017
 \end{array}$$

$$\begin{array}{rcl}
 A & \rightarrow & 10 * 4,096 = 40,960 \\
 F & \rightarrow & 15 * 256 = 3,840 \\
 1 & \rightarrow & 1 * 16 = 16 \\
 C & \rightarrow & 12 * 1 = 12 \\
 \hline
 AF1Ch & & = 44,828
 \end{array}$$

$$\begin{array}{rcl}
 3 & \rightarrow & 3 * 65,536 = 196,608 \\
 B & \rightarrow & 11 * 4,096 = 45,056 \\
 8 & \rightarrow & 8 * 256 = 2,048 \\
 D & \rightarrow & 13 * 16 = 208 \\
 2 & \rightarrow & 2 * 1 = 2 \\
 \hline
 3B8D2h & & = 243,922
 \end{array}$$

Afb. 1-3. Meer omzettingen van hexadecimaal naar decimaal.

Nu gaan we eens uitzoeken wat er gebeurt als we hex-getallen van meer dan een cijfer bij elkaar optellen. Hiervoor gebruiken we Debug en de getallen 3A7h en 1EDh:

```
-H 3A7 1ED
0594 01BA
```

We zien dat $3A7h + 1EDh = 594h$. U kunt deze uitkomst controleren door deze getallen in decimalen om te zetten en de som (of het verschil, zo u wilt) in decimale vorm te berekenen; bent u nog avontuurlijker aangelegd, berekenen ze dan rechtstreeks in hex.

$\begin{array}{r} 1 \\ 3A7 \\ + 92A \\ \hline CD1 \end{array}$	$\begin{array}{r} 1 \\ F451 \\ + CB03 \\ \hline 1BF54 \end{array}$	$\begin{array}{r} 1 \\ C \\ + D \\ \hline 19 \end{array}$
--	--	---

$\begin{array}{r} 1111 \\ BCD8 \\ + FAE9 \\ \hline 1B7C1 \end{array}$	$\begin{array}{r} 11 \\ BCD8 \\ + 0509 \\ \hline C1E1 \end{array}$
---	--

Afb. 1-4. Meer voorbeelden van hexadecimaal optellen.

1.5 Hex-getallen van vijf cijfers

Tot dusver is het rekenen in hex vrij eenvoudig. Wat gebeurt er nu wanneer we nog grotere getallen bij elkaar proberen op te tellen? Laten we eens een hex-getal van vijf cijfers proberen:

```
-H 5C3F0 4BC6
      ^ Error
```

Dat is onverwacht. Waarom zegt Debug dat dit fout is? Dat heeft te maken met een opslageenheid genaamd *woord*. De H-opdracht van Debug werkt alleen met woorden, en woorden zijn toevallig lang genoeg om vier hex-cijfers te bevatten, niet meer. Enkele bladzijden verderop leren we nog meer over woorden, maar onthoud voorlopig dat u met maar vier hex-cijfers kunt werken. Als u dus probeert twee hex-getallen van vier cijfers op te tellen, zoals C000h en D000h (wat een uitkomst van 19000h zou moeten geven), dan krijgt u in plaats daarvan:

-H C000 D000
9000 F000

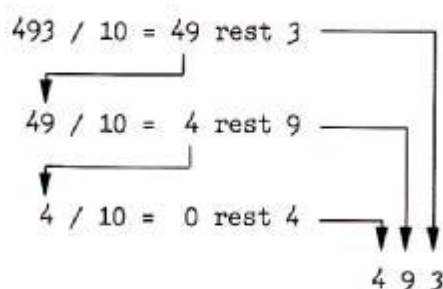
Debug houdt slechts de vier meest rechtse cijfers van de uitkomst vast.

1.6 Decimaal omzetten in hex

Tot dusver hebben we alleen naar de omzetting van hex naar decimaal gekeken. We gaan nu zien hoe decimale getallen naar hex worden omgezet. Zoals gezegd, zullen we in hoofdstuk 10 een programma schrijven waarmee de getallen van de 8088 als decimale getallen worden afgebeeld; in hoofdstuk 23 schrijven we een ander programma waarmee decimale getallen in de 8088 worden gelezen. Maar laten we, net als bij het omzetten van decimaal in hex, eens beginnen met te leren hoe je die omzettingen op papier doet. We beginnen weer met een lagere school-som.

Toen we leerden delen, deelden we 9 door 2 en kregen dan 4 rest 1. Die rest gaan we gebruiken om decimale getallen in hex-getallen om te zetten.

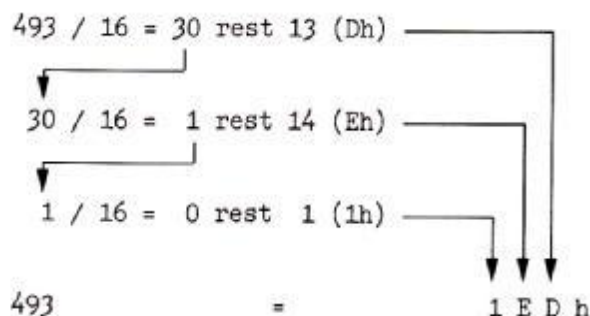
Laten we eens kijken wat er gebeurt wanneer we een decimaal getal herhaaldelijk delen — in dit geval 493 door tien:



De cijfers van het getal 493 komen in omgekeerde volgorde als de rest te voorschijn — dat wil zeggen, vanaf het meeste rechtse cijfer (3). In het vorige deel zagen we dat je voor de omzetting van hex naar decimaal alleen maar de machten van 10 hoeft te vervangen door de machten van 16.

Kunnen we nu bij de omzetting van decimaal naar hex delen door 16 in plaats van door 10? Ja, dat is de methode die daarbij wordt gebruikt.

Laten we bijvoorbeeld eens het hex-getal bij 493 opzoeken. We delen het als volgt door 16:



$$1069 / 16 = 66$$

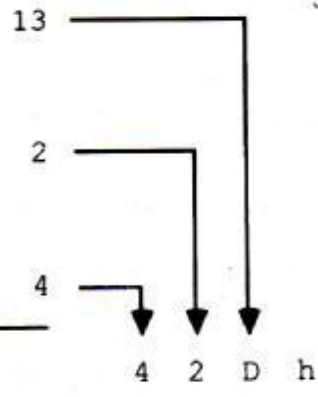


$$66 / 16 = 4$$



$$4 / 16 = 0$$

$$1069 =$$



$$57,109 / 16 = 3,569$$



$$3,569 / 16 = 223$$

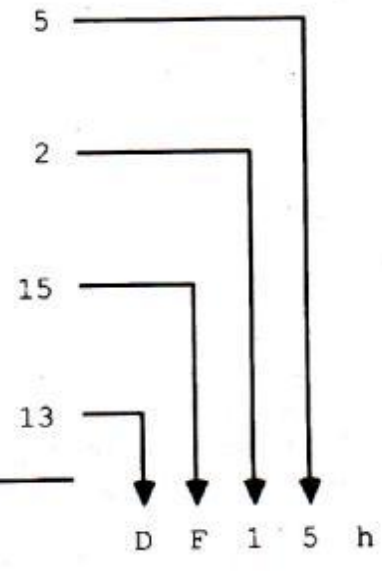


$$223 / 16 = 13$$



$$13 / 16 = 0$$

$$57,109 =$$



Afb. 1-5. Meer voorbeelden van omzettingen naar hexadecimaal.

We zien dat 1EDh het hex-equivalent van decimaal 493 is. Met andere woorden, blijf delen door 16, en vorm het uiteindelijke hex-getal door middel van de resten. Dat is alles.

1.7 Negatieve getallen

Zoals u zich nog zult herinneren, hebben we nog een vraag onbeantwoord gelaten bij het getal FFFFh. We zeiden dat FFFFh in feite -1 is. Maar als we FFFFh omzetten naar decimaal, krijgen we 65535. Hoe kan dat? Gedraagt het zich als een negatief getal?

Het zit zo. Als we FFFFh (ofwel -1) optellen bij 5, moet de uitkomst 4 zijn, omdat $5 - 1 = 4$. Gebeurt dat ook? Als we de H-opdracht van Debug gebruiken om 5 en FFFFh op te tellen, krijgen we:

```
-H 5 FFFF
0004 0006
-
```

Het *lijkt* of Debug FFFFh als -1 behandelt. Maar FFFFh zal zich niet altijd als -1 gedragen in programma's die we schrijven. Om te zien waarom niet, doen we een optelling op papier.

Als we twee decimale getallen optellen, zien we altijd dat er een naar de volgende kolom wordt overgedragen, zo:

```
  1 1
   9 5
+  5 8
-----
  1 5 3
```

Bij het optellen van twee hex-getallen gaat het niet veel anders. Als je 3 optelt bij F, krijg je 2, met een onthouden en overdragen naar de volgende kolom:

```
  1
   F
+  3
-----
  1 2 h
```

Kijk nu wat er gebeurt als we 5 optellen bij FFFFh:

```
  1 1 1 1
   0 0 0 5 h
+  F F F F h
-----
  1 0 0 4 h
```

Omdat $Fh + 1h = 10h$, zetten de successievelijke overdrachten keurig helemaal links een 1. En, als we die 1 negeren, hebben we het juiste antwoord op $5 - 1$, namelijk 4. Het mag vreemd lijken, maar FFFFh gedraagt zich als -1 wanneer we deze over-

loop negeren. Het wordt *overflow* genoemd omdat het getal nu vijf cijfers lang is, maar Debug alleen de eerste vier (meest rechtse) cijfers vasthoudt.

Is dat een fout of is dit antwoord juist? Ja en nee. We kunnen beide antwoorden kiezen. Spreken de uitkomsten elkaar dan niet tegen? Niet echt, want we kunnen deze getallen op twee manieren bekijken.

Zeg dat we FFFFh als gelijk aan 65535 beschouwen. Dat is een positief getal, en toevallig ook het grootste getal dat we met vier cijfers kunnen schrijven. We zeggen dat FFFFh een getal *zonder teken* is. Het heeft geen teken omdat we alle getallen van vier cijfers als positief hebben gedefinieerd. Als we 5 optellen bij FFFFh, krijgen we 10004h; de enige juiste uitkomst. In het geval van getallen zonder teken is een *overflow* dus fout.

We kunnen FFFFh echter ook behandelen als een negatief getal, zoals Debug deed toen we de H-opdracht gebruikten om FFFFh bij 5 op te tellen. FFFFh gedraagt zich als -1 zolang we de *overflow* negeren. De getallen 8000h tot en met FFFFh gedragen zich in feite allemaal keurig als negatieve getallen. Voor getallen *met teken*, zoals hier, is de *overflow* niet fout.

De 8088-microprocessor kan getallen als *zonder* of *met* teken beschouwen; de keus is aan u. De instructies ervoor verschillen ietwat, en we zullen die verschillen in hoofdstukken verderop nader onderzoeken wanneer we getallen voor de 8088 gaan gebruiken. Vóór u de negatieve waarde van, zeg, 3C8h, kunt leren schrijven, moeten we het begrip bit nader bekijken en zien hoe het past in de context van bytes, woorden en hexadecimaal.

1.8 Bits, bytes, woorden en binaire notatie

Het is tijd om ons te verdiepen in de fijne kneepjes van uw IBM-PC — tijd om te leren hoe de 8088 rekent: met binaire getallen. De 8088-microprocessor is met al zijn kracht nogal dom. Hij kent alleen de twee cijfers 0 en 1, dus elk getal dat hij gebruikt moet worden opgebouwd uit een reeks nullen en enen. Dit is het binaire talstelsel (grondtal 2).

Wanneer Debug een getal in hex afbeeldt, gebruikt hij een programmaatje om zijn interne getallen van binair om te zetten naar hexadecimaal. In hoofdstuk 5 zullen we een dergelijk programmaatje opzetten om binaire getallen in hexadecimale notatie weer te geven, maar eerst moeten we meer over binaire getallen zelf te weten komen.

Neem het binaire getal 1011b (de b staat voor binair). Dit getal is gelijk aan het decimale getal 11, ofwel Bh in hex. Om te zien hoe dat kan, vermenigvuldigen we de cijfers van 1011b met het grondtal van het getal, 2:

Machten van 2:

$$2^3 = 8$$

$$2^2 = 4$$

$$2^1 = 2$$

$$2^0 = 1$$

zo dat:

$$1 * 8 = 8$$

$$0 * 4 = 0$$

$$1 * 2 = 2$$

$$1 * 1 = 1$$

$$1011b = 11 \text{ of } Bh$$

Evenzo is 1111b gelijk aan Fh, of 15. En 1111b is het grootste binaire getal zonder teken van vier cijfers dat we kunnen schrijven, terwijl 0000b het kleinste is. Op die manier kunnen we met vier binaire cijfers 16 verschillende getallen schrijven. Er zijn precies 16 hex-cijfers, dus met elke vier binaire cijfers kunnen we een hex-cijfer schrijven.

Een hex-getal van twee cijfers, zoals 4Ch, kan worden geschreven als 0100 1100b. Het bestaat uit acht cijfers, die we in twee groepen van vier verdelen om ze gemakkelijker te kunnen lezen. Elk van deze binaire cijfers staat bekend als een *bit* (binary digit), dus een getal als 0100 1100b, ofwel 4Ch, is acht bits lang.

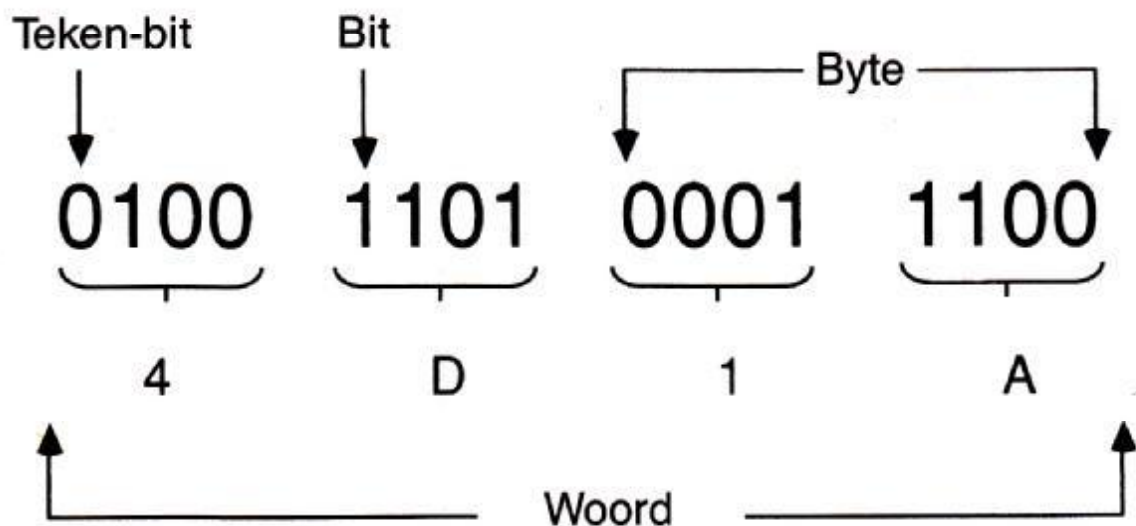
Heel vaak vinden we het gemakkelijk om elk van deze bits in een lange reeks te nummeren, met bit 0 helemaal rechts. De 1 in 10b is dan bit nummer 1, en het meest linkse bit in 1011b is bit nummer 3. Bits die op deze manier genummerd zijn, maken het ons gemakkelijker om over een bepaald bit te praten, zoals we verderop van plan zijn.

<u>Binair</u>	<u>Decimaal</u>	<u>Hexadecimaal</u>
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Afb. 1-6. Binair, hex en decimaal voor 0 t/m F.

Een groep van acht binaire cijfers wordt een *byte* genoemd, terwijl een groep van 16 binaire cijfers, of twee bytes, een *woord* is. We zullen deze termen het hele boek door veelvuldig gebruiken, omdat bits, bytes en woorden alle drie elementaire begrippen voor de 8088 vormen.

We zien nu ook waarom de hexadecimale notatie zo gemakkelijk is; in één byte passen precies twee hex-cijfers (vier bits per hex-cijfer), en in één woord passen precies vier hex-cijfers. Dat kunnen we niet zeggen van decimale getallen. Als we proberen voor een byte twee decimale cijfers te gebruiken, kunnen we geen getallen groter dan 99 schrijven, zodat we de waarden van 100 tot 255 kwijt zijn — meer dan de helft van het bereik van de getallen die een byte kan bevatten. En als we drie decimale cijfers gebruiken, moeten we meer dan de helft van de decimale getallen van drie cijfers negeren omdat de getallen 256 tot en met 999 niet in één byte kunnen.



Afb. 1-7. Een woord bestaat uit bits en bytes.

1.9 Twee-complement — een raar soort negatief getal

U weet nu voldoende om meer over negatieve getallen te leren. We zeiden al dat de getallen 8000h tot en met FFFFh zich allemaal als negatieve getallen gedragen wanneer we de overloop negeren. Er is een gemakkelijke manier om negatieve getallen op te sporen wanneer we ze in binair schrijven:

Positieve getallen:

0000h	0000 0000 0000 0000b
.	.
.	.
7FFFh	0111 1111 1111 1111b

Negatieve getallen:

8000h	1000 0000 0000 0000b
.	.
.	.
FFFFh	1111 1111 1111 1111b

In de binaire vorm is bij alle positieve getallen het eerste bit (bit 15) altijd 0. Bij alle negatieve getallen is dit eerste bit altijd 1. Dit verschil is in feite de manier waarop de 8088-microprocessor weet of een getal negatief is: hij kijkt naar bit 15, het *teken-bit*. Als we instructies voor getallen zonder teken in onze programma's gebruiken, zal de 8088 het teken-bit negeren, en kunnen we getallen met teken gebruiken zoals het ons uitkomt.

Die negatieve getallen staan bekend als het *twee-complement* van positieve getallen. Waarom complement? Omdat de omzetting van een positief getal zoals 3C8h naar zijn twee-complement-vorm een proces van twee stappen is, waarvan het eerste de omzetting inhoudt van het getal naar zijn *complement*.

We hoeven getallen niet vaak negatief te maken, maar we doen de omzetting hier even om te laten zien hoe de 8088-microprocessor getallen negatief maakt. De omzetting zal u wat vreemd voorkomen. U zult niet zien waarom het werkt, maar wel hoe het werkt.

Om het twee-complement (de negatieve waarde) van een getal te vinden, schrijven we eerst het getal in binair, zonder op het teken te letten. Bijvoorbeeld 4Ch wordt dan 0000 0000 0100 1100b.

Om dit getal negatief te maken, keren we eerst alle nullen en enen om. Dit proces van omkering heet *complementeren*, en als we het complement van 4Ch nemen, hebben we:

0000 0000 0100 1100

en dat wordt:

1111 1111 1011 0011

Bij de tweede omzetting-stap tellen we er 1 bij op:

	11	
1111 1111 1011 0011		
+	1	
1111 1111 1011 0100		
	-4Ch = FFB4h	

Het antwoord, FFB4h, is de uitkomst die we krijgen als we de H-opdracht van Debug gebruiken om 4Ch van 0 af te trekken.

Als u wilt, kunt u FFB4h en 4C op papier optellen om te controleren of het antwoord 10000h is. En van onze eerdere bespreking weet u dat u die 1 helemaal links bij de twee-complement-optelling moet negeren om 0 te krijgen ($4C + (-4C) = 0$).

1.10 Samenvatting

We hebben in dit hoofdstuk een flinke klim gemaakt naar de wereld van hexadecimale en binaire getallen, en u hebt zich daar misschien aardig bij moeten inspannen. Straks, in hoofdstuk 3, zullen we het tempo wat verlagen — als u voldoende weet om in hex met Debug overweg te kunnen. We halen nu even diep adem en kijken achterom naar waar we geweest zijn en wat we hebben gezien.

We ontmoetten eerst Debug. In komende hoofdstukken zullen we Debug heel goed leren kennen, maar omdat hij onze vertrouwde decimale getallen niet begrijpt, zijn we de vriendschap begonnen met het leren van een nieuw talstelsel, de hexadecimale notatie.

Bij het leren over hex-getallen leerde u ook decimale getallen omzetten naar hex, en hex-getallen naar decimale. We zullen verderop nog een programma schrijven om die vertalingen tot stand te brengen, maar het was nu nodig om de taal zelf te leren. Toen we eenmaal de grondbeginselen van de hexadecimale notatie hadden gehad, konden we een zijspoor volgen in de richting van bits, bytes, woorden en binaire getallen — belangrijke zaken die u vaak zult tegenkomen op uw onderzoekingsstocht door de wereld van de 8088 en het programmeren in assembleertaal.

Ten slotte vervolgden we onze tocht met een bespreking van negatieve getallen in hex — de twee-complement-getallen. Dat bracht ons op de getallen met en zonder teken, waarbij we twee soorten overflow zagen: een waarbij een overflow het juiste antwoord geeft (optellen van twee getallen met teken), en een waarbij de overflow het verkeerde antwoord geeft (optellen van twee getallen zonder teken).

Al dat leren zal in verdere hoofdstukken de moeite waard blijken te zijn, omdat we onze kennis van hex-getallen zullen gebruiken om te praten met Debug, en Debug als tolk zal fungeren tussen ons en de 8088-microprocessor binnenin de IBM-PC. In het volgende hoofdstuk gaan we de kennis gebruiken die we tot dusver hebben vergaard om meer over de 8088 te weten te komen. We zullen ons daarbij weer verlaten op Debug, en hex-getallen in plaats van binaire getallen gebruiken om met de 8088 te praten. We zullen leren over de registers van de microprocessor — de plaatsen waar hij getallen opslaat — en in hoofdstuk 3 zullen we voldoende weten om een echt programma te schrijven dat een teken op het scherm afdrukt. We zullen ook meer leren over hoe de 8088 rekent; in hoofdstuk 10 gaan we een programma schrijven dat binaire getallen omzet in decimale.

2 Rekenen en de 8088

- 2.1 Registers als variabelen 34**
- 2.2 Geheugen en de 8088 35**
- 2.3 Optellen volgens de 8088 37**
- 2.4 Aftrekken volgens de 8088 39**
- 2.5 Negatieve getallen in de 8088 40**
- 2.6 Bytes in de 8088 40**
- 2.7 Vermenigvuldigen en delen volgens de 8088 41**
- 2.8 Samenvatting 44**

Nu we iets weten van Debug en zijn hex-getallen en de 8088 en zijn binaire getallen, kunnen we leren hoe de 8088 zijn rekenkundige bewerkingen uitvoert door middel van interne commando's, de *instructies*.

2.1 Registers als variabelen

Debug, onze gids en tolk, weet veel over de 8088-microprocessor binnen de IBM-PC. We zullen hem gebruiken om uit te zoeken hoe de 8088 van binnen werkt, en beginnen met Debug te vragen wat hij kan laten zien van kleine stukjes geheugen genaamd *registers*, waarin we getallen kunnen opslaan. Registers lijken op de variabelen in BASIC, maar zijn niet helemaal hetzelfde. De 8088-microprocessor bevat een vast aantal registers, en deze registers maken geen deel uit van het geheugen van uw IBM-PC. We zullen Debug vragen de registers van de 8088 weer te geven op ons beeldscherm met de opdracht R (van *Register*):

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

(U zult waarschijnlijk andere getallen op de tweede en derde regel van uw scherm zien; die getallen weerspiegelen de hoeveelheid geheugen in een computer. U zult zulke verschillen blijven zien, en verderop zullen we er meer over vertellen.)

Voorlopig heeft Debug ons een hoop informatie verschaft. Laten we eerst eens de eerste vier registers, AX, BX, CX en DX, bekijken, die volgens Debug allemaal 0000 zijn, zowel hier als op uw scherm. Dit zijn de *algemene* registers. De andere registers, SP, BP, SI, DI, DS, ES, SS, CS en IP, zijn speciale registers die we in volgende hoofdstukken zullen behandelen.

Het getal van vier cijfers na elke registernaam staat in hex. In hoofdstuk 1.1 hebben we geleerd dat een woord precies met vier hex-cijfers wordt omschreven. Hier kunt u zien dat elk van de 13 registers van de 8088 een woord, of 16 bits, lang is. Daarom staan computers die zijn gebaseerd op de 8088-microprocessor ook bekend als 16-bits machines.

We zeiden al dat de registers iets weg hebben van BASIC-variabelen. Dat betekent dat we ze zouden moeten kunnen veranderen, en dat kan ook. De R-opdracht van Debug doet meer dan alleen registers laten zien. Als je de naam van het register erachter zet, zegt de opdracht tegen Debug dat we het register willen zien, en dan veranderen. We kunnen bijvoorbeeld het AX-register als volgt wijzigen:

```
-R AX
AX 0000
:3A7
-
```

Laten we eens kijken of het AX-register nu 3A7h bevat:

-R

```
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP EI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

Ja dus. We kunnen met de R-opdracht verder elk hex-getal in elk register zetten door de naam van het register op te geven en na de dubbele punt het nieuwe getal te tikken, zoals we net deden. Van nu af zullen we deze opdracht gebruiken wanneer we een getal in een van de registers van de 8088 willen zetten.

U zult zich nog het getal 3A7h nog herinneren van hoofdstuk 1, waar we de H-opdracht van Debug gebruikten om 3A7h en 1EDh op te tellen. Daar deed Debug het werk voor ons. Deze keer zullen we Debug alleen als tolk gebruiken om rechtstreeks met de 8088 te kunnen werken. We zullen de 8088 instructies geven om getallen uit twee registers op te tellen. Daartoe zetten we een getal in het BX-register en geven dan de 8088 opdracht om het getal in BX op te tellen bij het getal in AX en het antwoord weer terug in AX te zetten. Eerst hebben we een getal in het BX-register nodig. Laten we nu eens 3A7h en 92Ah optellen. Gebruik de R-opdracht om 92Ah in BX op te slaan.

2.2 Geheugen en de 8088

De registers AX en BX moeten nu respectievelijk 3A7h en 92Ah bevatten, zoals we met de R-opdracht kunnen nagaan:

```
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

Hoe moeten we de 8088, nu we onze twee getallen in de registers AX en BX hebben, vertellen dat hij BX bij AX moet optellen? We zetten enkele getallen in het geheugen van de computer.

Uw IBM-PC heeft waarschijnlijk minstens 128K geheugen — veel meer dan we hier nodig hebben. We zullen twee bytes *machinecode* ergens in een hoekje van die enorme hoeveelheid geheugen zetten. In dit geval zal de machinecode bestaan uit twee binaire getallen die de 8088 opdragen het BX-register op te tellen bij het AX-register. Om te zien wat er gebeurt, zullen we daarna deze instructie *uitvoeren* met behulp van Debug.

Waar moeten we nu onze instructie van twee bytes in het geheugen zetten, en hoe vertellen we de 8088 waar hij haar kan vinden? Het blijkt dat de 8088 geheugen in stukken van 64K hakt, die bekend staan als *segmenten*. Meestal zullen we het geheugen binnen één van die segmenten bekijken zonder echt te weten waar het segment begint. Dat is mogelijk door de manier waarop de 8088 geheugen aanduidt.

Alle bytes in het geheugen worden aangeduid met getallen, vanaf 0h en hoger. Maar weet u nog dat hex-getallen uit niet meer dan vier cijfers konden bestaan? Dat betekent dat het grootste getal dat de 8088 kan gebruiken het hex-equivalent van 65635

is, wat betekent dat het maximum aantal adressen dat het kan gebruiken, 64K beslaat. Maar we weten uit ervaring dat de 8088 méér dan 64K geheugen kan aanspreken. Hoe doet hij dat dan? Door een klein trucje toe te passen: hij gebruikt twee getallen, één voor elk segment van 64K, en één voor elke byte, of *offset*, binnen het segment. Elk segment begint op een veelvoud van 16 bytes, dus door segmenten en offsets te overlappen kan de 8088 in feite meer dan 64K geheugen benoemen. Dit is precies de manier waarop de 8088 tot wel een miljoen bytes geheugen gebruikt.

Alle adressen (labels) die we zullen gebruiken, worden gerekend vanaf het begin van een segment; de afstand ertussen wordt *offset* genoemd. We zullen adressen schrijven als een segmentnummer gevolgd door de offset binnen het segment. Zo betekent bijvoorbeeld 3765:0100 een offset 100h binnen segment 3765.

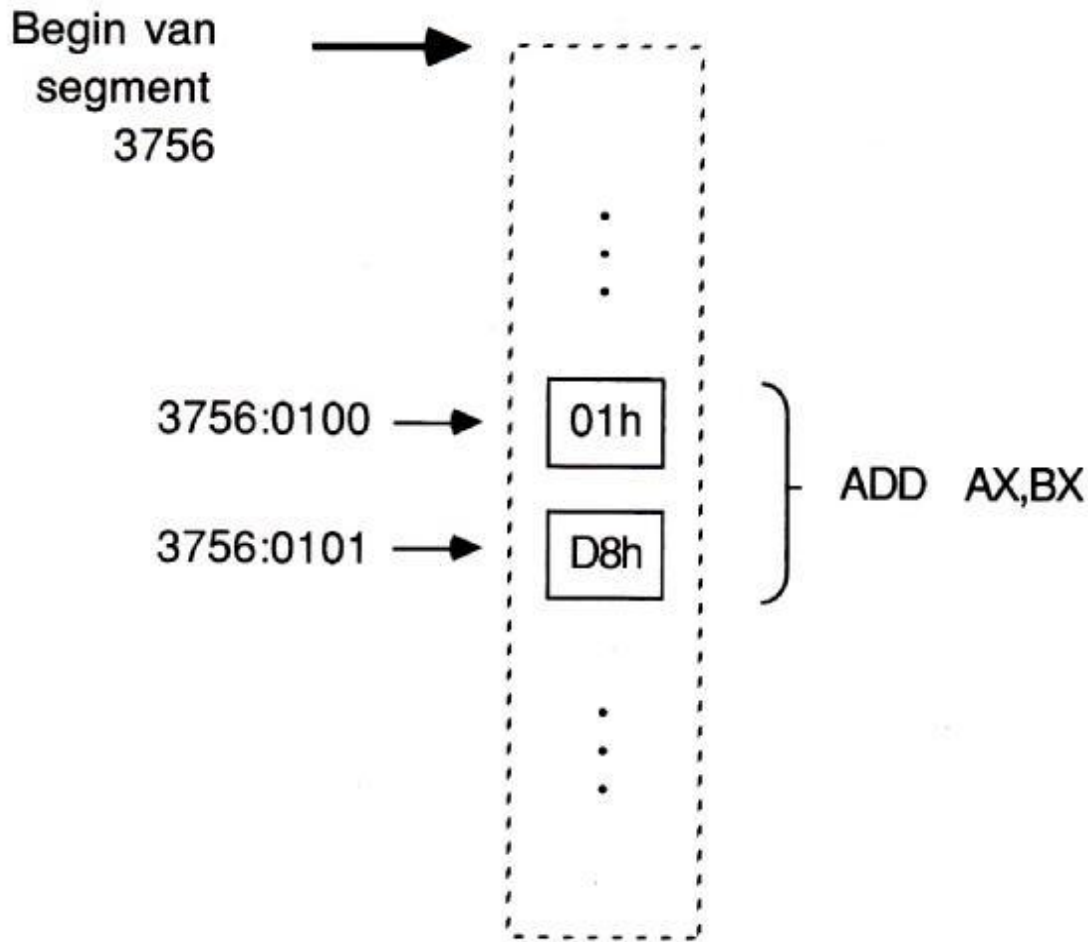
Later, in hoofdstuk 11, leren we meer over segmenten en zullen we zien waarom het segmentnummer zo hoog is. Maar laten we er voorlopig van uitgaan dat Debug voor de segmenten zorgt, zodat wij met een segment kunnen werken zonder op segmentnummers te hoeven letten. En voorlopig zullen we adressen alleen met hun offset aanduiden. Elk van deze adressen slaat op slechts één byte in het segment, en de adressen komen na elkaar, zodat 101h de byte ná 100h in het geheugen is.

Als we haar uitschrijven, ziet onze twee bytes lange instructie voor het optellen van BX en AX er zo uit: ADD AX,BX. We gaan deze instructie op de plaatsen 100h en 101h zetten, in het segment waarmee Debug begint. Als we het over onze ADD (tel op)-instructie hebben, zeggen we dat ze op plaats 100h staat, omdat dat de plaats van de eerste byte van de instructie is.

De Debug-opdracht voor het onderzoeken en wijzigen van geheugen is E, van *Enter* (invoeren). Gebruik deze opdracht om de twee bytes van de ADD-instructie als volgt in te voeren:

```
-E 100
3756:0100 E4.01
-E 101
3756:0101 85.D8
-
```

De getallen 01h en D8h zijn de machinetaal-vorm van de 8088 voor onze ADD-instructie op de geheugenplaatsen 3756:0100 en 3756:0101. Het segmentnummer dat u ziet is waarschijnlijk anders, maar dat verschil is niet van invloed op ons programma. Ook zal Debug waarschijnlijk een ander getal van twee cijfers hebben afgebeeld bij uw beide E-opdrachten. Deze getallen (E4h en 85h in ons voorbeeld) zijn de oude getallen in het geheugen op de offset-adressen 100h en 101h van het segment dat Debug heeft gekozen — dat wil zeggen, de getallen zijn gegevens uit vorige programma's die in het geheugen zijn achtergebleven toen u Debug startte. (Als u uw computer net hebt aangezet, moeten de getallen 00 zijn.)



Afb. 2-1. Onze instructie begint 100h bytes vanaf het begin van het segment.

2.3 Optellen volgens de 8088

Op dit moment moet er nu het volgende op uw scherm staan:

```
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 01D8          ADD     AX,BX
```

Onze ADD-instructie is keurig in het geheugen gezet, precies waar we haar wilden hebben. Dat is te zien op de derde regel. De eerste twee getallen, 3756:0100, geven ons het adres (100h) van het eerste getal van onze ADD-instructie. Daarnaast zien we de twee bytes voor ADD: 01D8. De byte gelijk aan 01h staat op adres 100h, terwijl D8h op 101h staat. Ten slotte bevestigt de melding ADD AX,BX dat we de instructie juist hebben ingevoerd. Dat is prettig omdat we immers onze instructie in *machine-taal* hebben gegeven — getallen die geen betekenis voor ons hebben maar die de 8088 uitlegt als een optel-instructie.

We hebben nu onze ADD-instructie in het geheugen gezet, maar zijn nog niet zover

dat we haar door de 8088 kunnen laten draaien (haar *uitvoeren*). Eerst moeten we de 8088 vertellen wáár hij de instructie kan vinden.

De 8088 zoekt segment- en offset-adressen op in twee speciale registers, CS en IP, die u in de register-afbeelding hierboven kunt zien staan. Het segmentnummer staat in het CS- of *Code Segment*-register, dat we straks zullen bespreken. Als u de register-weergave bekijkt, kunt u zien dat Debug het CS-register al voor ons heeft ingesteld (CS=3756 in ons voorbeeld). Het volledige beginadres van onze instructie is echter 3756:0100.

Het tweede deel van dit adres (de offset binnen segment 3756) is opgeslagen in het IP (*Instruction Pointer*)-register. De 8088 gebruikt de offset in het IP-register om onze eerste instructie echt te kunnen vinden. We kunnen hem zeggen waar hij moet kijken door het IP-register in te stellen op het adres van onze eerste instructie — IP = 0100.

Maar het IP-register is al ingesteld op 100h. We zijn slim geweest: Debug stelt IP altijd op 100h in wanneer u hem de eerste keer start. Omdat we dat wisten, hebben we expres 100h als het adres van onze eerste instructie gekozen en op die manier voorkomen dat het IP-register afzonderlijk moest worden ingesteld. Het is van belang dat u dit goed onthoudt.

Nu onze instructies op hun plaats staan en de registers goed zijn ingesteld, zullen we Debug opdragen onze ene instructie uit te voeren. We gebruiken daarvoor de Debug-opdracht T (van *Trace*), die één instructie per keer uitvoert en dan de registers laat zien. Na elke stap dient IP naar de volgende instructie te wijzen. In dit geval zal het wijzen naar 102h. We hebben geen instructie op 102h gezet, dus op de laatste regel van de register-afbeelding zullen we slechts een instructie zien die van een ander programma is overgebleven.

We zullen Debug eens vragen om met de T-opdracht een instructie te traceren:

```
-T
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0102  NV UP DI PL NZ AC PE NC
3756:0102 AC          LODSB
```

Dat is 't. In het AX-register staat nu CD1h, wat de som is van 3A7h en 92Ah. En het IP-register wijst naar adres 102h, dus de laatste regel van de register-afbeelding laat een instructie op het geheugenadres 102h, en niet 100h, zien.

We zeiden al dat de instructiewijzer (IP) samen met het CS-register altijd naar de volgende instructie voor de 8088 wijst. Als we weer T zouden tikken, zouden we de volgende instructie uitvoeren, maar doe dat nu nog niet — uw 8088 zou totaal van streek kunnen raken.

Als we nu in plaats daarvan onze ADD-instructie nogmaals willen uitvoeren, en 92Ah willen optellen bij CD1h en de nieuwe uitkomst opslaan in AX? Daartoe moeten we de 8088 vertellen waar hij de volgende instructie kan vinden, en we willen dat dat onze ADD-instructie op 0100h is. Kunnen we nu gewoon het IP-register veranderen in 0100h? Laten we het eens proberen. Gebruik de R-opdracht om IP op 100 te zetten en bekijk de register-afbeelding:


```

AX=0CD1  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100  NV UP DI PL NZ AC PE NC
3756:0100 01D8          ADD     AX,BX

```

Dat werkt dus. Probeer de T-opdracht nu nog eens en kijk of er 15FBh in het AX-register staat. Klopt ook.

Zoals u kunt zien, moet u altijd het IP-register en de instructie onderaan een register-afbeelding nagaan voor u de T-opdracht geeft. Op die manier weet u zeker dat de 8088 de instructie uitvoert die u wilt.

Zet nu het IP-register weer op 100h, kijk goed of u AX = 15FB en BX = 092A ziet, en laten we eens kijken of we kunnen aftrekken.

AX: 03A7 BX: 092A

 **ADD AX,BX**
LODSB

Afb. 2-2. Voor uitvoering van de ADD-instructie.

AX: 0CD1 BX: 092A

 **ADD AX,BX**
LODSB

Afb. 2-3. Na uitvoering van de ADD-instructie.

2.4 Aftrekken volgens de 8088

We gaan een instructie schrijven om BX zodanig van AX af te trekken dat we na twee aftrekkingen 3A7h in AX hebben: het punt waarop we voor onze twee optellingen begonnen. U zult ook zien hoe we bij het invoeren van twee bytes in het geheugen wat werk kunnen besparen.

Toen we de twee bytes voor onze ADD-instructie invoerden, tikten we de E-opdracht twee keer: een keer met 0100h voor het eerste adres, en een keer met 0101h voor het tweede adres. Dit ging goed, maar het blijkt dat we de tweede byte eigenlijk zonder een extra R-opdracht kunnen invoeren als we hem door een spatie van de eerste byte scheiden. Als u de bytes hebt ingevoerd, wordt door een druk op de Enter-toets de E(nter)-opdracht weer beëindigd. Probeer deze methode eens bij onze aftrek-instructie:

```
-E 100  
3765:0100 01.29 D8.D8
```

De register-afbeelding (denk eraan het IP-register weer op 100h te zetten) dient nu de instructie *SUB AX,BX* te bevatten, waardoor het BX-register wordt afgetrokken van het AX-register en de uitkomst in AX komt te staan. Het lijkt misschien dat AX en BX in omgekeerde volgorde staan, maar de instructie is net als de BASIC-opdracht $AX = AX - BX$, behalve dat de 8088, anders dan BASIC, altijd het antwoord in de eerste variabele (register) zet.

Voer deze instructie uit met de T-opdracht. In AX moet nu CD1 staan. Wijzig IP zo dat hij terugwijst naar deze instructie en voer haar nogmaals uit (denk eraan dat u eerst de instructie onderaan de R-afbeelding controleert). AX moet nu 03A7 zijn.

2.5 Negatieve getallen in de 8088

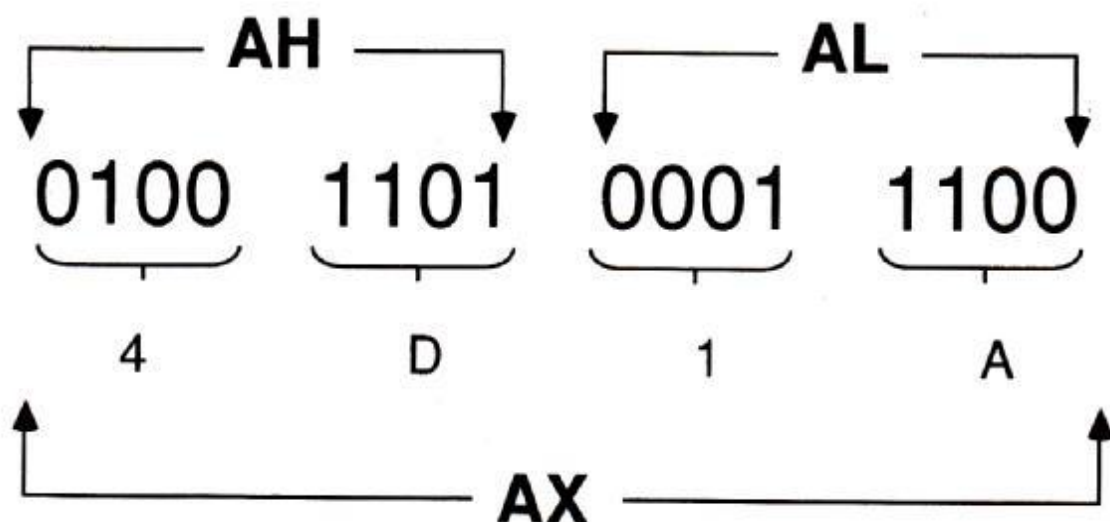
In het vorige hoofdstuk zagen we hoe de 8088 voor negatieve getallen de twee-complement-vorm gebruikt. Laten we eens rechtstreeks de SUB-instructie gebruiken om negatieve getallen te berekenen. We zullen de 8088 testen om te zien of we FFFFh voor -1 krijgen. Daartoe trekken een van nul af en, als het klopt wat we denken, komt er dan FFFFh (-1) in AX te staan. Zet AX op nul en BX op een, kijk met de T-opdracht op adres 0100h wat de instructie heeft gedaan. Precies wat we dachten: $AX = FFFFh$

Nu u deze aftrek-instructie onder de knie hebt, hebt u misschien zin om nog wat meer getallen te proberen om meer feeling te krijgen voor het rekenen met twee-complementen. Probeer bijvoorbeeld eens wat u krijgt bij -2 .

2.6 Bytes in de 8088

We hebben al ons rekenwerk tot dusver gedaan met woorden, vandaar de vier hex-cijfers. Kan de microprocessor nu ook met bytes rekenen? Ja.

Omdat een woord uit twee bytes bestaat, kan elk algemeen register worden verdeeld in twee bytes, die bekend staan als de *hoge byte* (de eerste twee hex-cijfers) en de *lage byte* (de tweede twee hex-cijfers). Elk van deze registers kan worden aangeropen met zijn letter (A tot en met D), gevolgd door X voor een woord, H voor de hoge byte of L voor de lage byte. Zo zijn DL en DH bijvoorbeeld byte-registers, en is DX een woord-register. (Deze termen echter kunnen wat verwarring scheppen omdat woorden met eerst de lage byte en daarna de hoge byte in het geheugen worden opgeslagen.)



Afb. 2-4. Het AX-register gesplitst in twee byte-registers (AH en AL).

We gaan eens kijken hoe het rekenen met bytes bij een ADD-instructie werkt. Voer eerst de twee bytes 00h en C4h in, vanaf geheugenplaats 0100h. Onderaan in de register-afbeelding ziet u dan de instructie *ADD AH,AL*, waardoor de twee bytes van het AX-register worden opgeteld en de uitkomst in de hoge byte, AH, wordt gezet.

Zet vervolgens 0102h in het AX-register. Hierdoor komt er in het AH-register 01h en in het AL-register 02h te staan. Zet het IP-register op 100h, voer de T-opdracht uit en u zult zien dat er in AX nu 0302 staat. De uitkomst van 01h + 02h is 03h, en die waarde staat in het AH-register.

Maar stel dat u niet 01h en 02h hadden willen optellen. Stel dat u eigenlijk 01h en 03h wilde optellen. Als er in het AX-register al 0102 stond, zou u dan het AL-register in 03h kunnen veranderen? Nee. U zou het AX-register in 0103h moeten veranderen. Waarom? Omdat Debug ons alleen *hele* woord-registers laat veranderen. We kunnen met Debug niet alleen het lage of het hoge deel van een register wijzigen. Maar dat is, zoals u in het vorige hoofdstuk al zag, geen probleem. Met hex-getallen kunnen we een woord in twee bytes splitsen door het viercijferige hex-getal in tweeën te delen. Daardoor wordt het woord 0103h verdeeld in de twee bytes 01h en 03h.

Om deze ADD-instructie te proberen, moet u 0103h in het AX-register zetten. Uw ADD AH,AL-instructie staat nog op de geheugenplaats 0100h, dus zet het IP-register weer op 100h en loop haar met de T-opdracht door met 01h en 03h nu in de registers AH en AL. In AX staat nu 0403h: 04h, de som van 01h + 03h staat in het AH-register.

2.7 Vermenigvuldigen en delen volgens de 8088

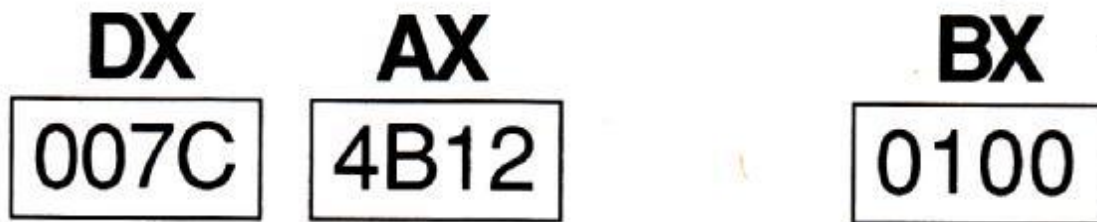
We hebben gezien hoe de 8088 twee getallen optelt en aftrekt. Nu gaan we zien dat hij ook kan vermenigvuldigen en delen — slimme processor die hij is. De vermenigvuldig-instructie wordt MUL (van *multiply*) genoemd, en de machinecode



Afb. 2-5. Voor uitvoering van de MUL-instructie.



Afb. 2-6. Na uitvoering van de MUL-instructie.



Afb. 2-7. Voor uitvoering van de DIV-instructie.



Afb. 2-8. Na uitvoering van de DIV-instructie.

voor het vermenigvuldigen van AX en BX is F7h E3h. We zullen dit in het geheugen invoeren, maar eerst iets over de MUL-instructie.

Waar slaat de MUL-instructie zijn antwoord op? In het AX-register? Niet helemaal; we moeten hier oppassen. Zoals u straks zult zien, kan het vermenigvuldigen van twee 16-bits getallen een uitkomst van 32 bits geven, dus de MUL-instructie slaat zijn uitkomst op in *twee* registers, DX en AX. De 16 hoogste bits worden in het DX-register gezet, de 16 laagste in AX. We kunnen deze register-combinatie ook schrijven als DX:AX.

We gaan nu terug naar Debug en 8088. Voer de vermenigvuldig-instructie, F7h E3h in, op de geheugenplaats 0100h, net zoals bij de optel- en aftrek-instructie, en maak $AX = 7C4Bh$ en $BX = 100h$. In de register-afbeelding zult u de instructie dan als *MUL BX* zien, zonder enige verwijzing naar het AX-register. Om woorden, zoals hier, te vermenigvuldigen, vermenigvuldigt de 8088 *altijd* het register dat u in de instructie opgeeft met het AX-register, en slaat hij het antwoord op in het registerpaar DX:AX.

Laten we, voor we de MUL-instructie echt uitvoeren, haar eens op papier doen. Hoe berekenen we $100h * 7C4Bh$? De drie cijfers 100 hebben in hex hetzelfde effect als in decimaal, dus om met 100h te vermenigvuldigen, zetten we er gewoon twee nullen aan de rechterkant van een hex-getal bij. Dus $100h * 7C4Bh = 7C4B00h$. Deze uitkomst is te lang voor een woord, dus we zullen haar in de twee woorden 007Ch en 4B00h splitsen.

Gebruik Debug om de instructie te traceren. U zult zien dat DX het woord 007Ch en AX het woord 4B00h bevat. Met andere woorden, de 8088 heeft de uitkomst van de vermenigvuldigde *woorden* in het registerpaar DX:AX gezet. Omdat het vermenigvuldigen van twee woorden nooit tot een uitkomst van meer dan twee woorden kan leiden, maar vaak langer dan een woord is (zoals we net hebben gezien), zet de woordvermenigvuldigings-instructie de uitkomst altijd in het registerpaar DX:AX. En delen? Als we getallen delen, houdt de 8088 zowel de uitkomst als de rest van de deling vast. Laten we eens kijken hoe de 8088 getallen deelt.

Zet eerst de instructie F7h F3h op 0100h (en 101h). DIV gebruikt net als de MUL-instructie DX:AX zonder dat hem dat wordt opgedragen, dus het enige wat we zien, is *DIV BX*. Laad nu de registers zodanig dat $DX = 007Ch$ en $AX = 4B12h$; in BX moet nog steeds 0100h staan.

We rekenen weer eerst de uitkomst op papier uit: $7C4B12h / 100H = 7C4Bh$ rest 12. Wanneer we onze deling-instructie nu uitvoeren, zien we dat $AX = 7C4Bh$, de uitkomst van onze deling, en $DX = 0012h$, de rest. (We zullen deze rest nog heel goed kunnen benutten in hoofdstuk 10, als we een programma schrijven dat decimale getallen omzet in hex door gebruik van de resten te maken, net zoals we in hoofdstuk 1 hebben gedaan.)

2.8 Samenvatting

Het is bijna zover dat we een echt programma kunnen schrijven — dat een teken op het scherm afbeeldt. We hebben onze tijd besteed aan het leren van de grondbeginnels. Laten we kijken hoe ver we zijn gekomen; dan zijn we daarna klaar om door te gaan.

We zijn dit hoofdstuk begonnen met te leren over registers en de overeenkomst die ze hebben met BASIC-variabelen. We zagen echter dat de 8088, anders dan BASIC, een klein en vast aantal registers heeft. We hebben ons op de vier algemene registers gericht, met snel een blik op de registers CS en IP, die de 8088 gebruikt om segment- en offset-adressen op te zoeken.

Na te hebben geleerd hoe registers kunnen worden gewijzigd en gelezen, gingen we verder met programmaatjes van één instructie lang, die we invoerden als machinecode en die twee getallen in de registers AX en BX optelden, aftrokken, vermenigvuldigden en deelden. In latere hoofdstukken zullen we veel gebruiken van wat we hier hebben geleerd, maar u hoeft de machinecode voor elke instructie niet te onthouden. We hebben ook geleerd hoe we Debug een enkele instructie moeten laten uitvoeren, of traceren. We zullen nog zeer op Debug steunen bij het schrijven van onze programma's. Naarmate onze programma's langer worden, zal het traceren van de instructies natuurlijk vervelender zowel als nuttiger worden. Later zullen we onze ervaring gebruiken om te leren hoe we meer dan één instructie met een Debug-opdracht kunnen uitvoeren.

Laten we nu teruggaan naar echte programma's en leren hoe je een programma maakt dat kan praten.

3 Tekens afbeelden

- 3.1 INT — de krachtige interrupt 47**
- 3.2 Een net einde — INT 20h 49**
- 3.3 Een programma van twee regels — de stukjes bij elkaar leggen 49**
- 3.4 Programma's invoeren 50**
- 3.5 Gegevens in registers zetten 51**
- 3.6 Een reeks tekens schrijven 53**
- 3.7 Samenvatting 55**

We weten nu voldoende om iets degelijks aan te pakken, dus stroop uw mouwen op en strek uw vingers. We beginnen met DOS opdracht te geven een teken naar het scherm te sturen en gaan dan verder met nog interessantere dingen. Eerst maken we daartoe een programmaatje met meer dan één instructie, en daarna leren we een andere manier om gegevens in registers te zetten — deze keer vanuit een programma. Maar eerst proberen we DOS aan de praat te krijgen.

3.1 INT — de krachtige interrupt

Aan onze vier rekeninstructies ADD, SUB, MUL en DIV voegen we een nieuwe instructie toe, genaamd INT (van *Interrupt*, of onderbreking). De INT-instructie heeft iets weg van de BASIC-opdracht GOSUB. We zullen INT gebruiken om DOS te vragen een teken, A, voor ons op het scherm af te beelden.

Voor we leren hoe INT werkt, bekijken we een voorbeeld. Start Debug en zet 200h in AX en 41h in DX. De INT-instructie voor DOS-functies is INT 21h, in machinecode CDh 21h. Net als de DIV-instructie van het vorige hoofdstuk is dit een instructie van twee bytes. Zet INT 21h in het geheugen vanaf 100h en gebruik de R-opdracht om na te gaan of de instructie inderdaad INT 21 is (vergeet niet IP op 100h te zetten als hij die waarde al niet heeft).

We zijn nu klaar om deze instructie uit te voeren, maar kunnen de traceer-opdracht (T) hier niet gebruiken zoals in het vorige hoofdstuk. De T-opdracht voert één instructie tegelijk uit, maar de INT-instructie roept een groot programma in DOS aan voor het eigenlijke werk, ongeveer zoals BASIC-programma's een subroutine kunnen aanroepen met de GOSUB-opdracht.

We willen niet alle instructies in de hele DOS-'subroutine' uitvoeren door ze stuk voor stuk te traceren. In plaats daarvan willen we ons programmaatje van een regel *draaien*, maar stoppen vóórdat de instructie op geheugenplaats 102h wordt uitgevoerd. Dat kan met de G (*Go*)-opdracht, gevolgd door het adres waar we willen stoppen:

-G 102

A

```
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC
3970:0102 8BE5          MOV     SP,BP
```

DOS heeft het teken A afgedrukt en daarna de besturing weer aan ons programma overgedragen. (Denk eraan: de instructie op 102 bestaat uit gegevens die door een ander programma zijn achtergelaten, dus u zult waarschijnlijk iets anders zien.)

Ons programmaatje hier bestaat in zekere zin uit twee instructies; waarbij de tweede instructie is wat er op geheugenplaats 102 staat. Dat wil zeggen, het ziet er ongeveer zo uit:

```
INT     21
MOV     SP,BP          ;(of wat er maar in uw computer zit)
```

We zullen deze willekeurige tweede instructie straks door een eigen instructie vervangen. Omdat er nu niets staat wat we uitgevoerd willen hebben, hebben we Debug ech-

ter voorlopig gezegd dat hij het programma moet draaien, met de uitvoering moet stoppen als hij die tweede instructie heeft bereikt en de registers moet afbeelden wanneer hij klaar is.

En hoe wist DOS nu hoe het die A moest afbeelden? De 02h in het AH-register vertelde DOS dat het een teken moest afdrukken. Een ander nummer in AH zou DOS hebben opgedragen een andere functie uit te voeren. (We zien verderop nog andere DOS-functies. Een volledige lijst van deze functies vindt u in appendix E. Wilt u nog vollediger geïnformeerd worden over INT 21h-functies, dan raad ik u aan een van de volgende boeken te raadplegen: *MS-DOS voor gevorderden* (Ray Duncan, Kluwer Technische Boeken 1987, ISBN 90 201 1992 3), of *Handboek voor IBM-programmeurs* (Peter Norton, Kluwer Technische Boeken 1987, ISBN 90 201 2054 9.)

Wat het teken zelf betreft: DOS gebruikt het getal in het DL-register als de ASCII-code voor het af te beelden teken wanneer we het vragen een teken naar het scherm te sturen. We hebben er 41h in opgeslagen, de ASCII-code voor de hoofdletter A. In appendix E vindt u een tabel met de ASCII-codes voor alle tekens die uw IBM-PC kan afbeelden. Voor uw gemak staan de getallen zowel in decimaal als in hex. Maar omdat Debug alleen hex kan lezen, is dit een mooie gelegenheid voor u om nog eens te oefenen in het omzetten van decimale naar hex-getallen. Kies een teken uit de tabel en zet die zelf om in hex. Controleer dan of uw omzetting klopt door uw hex-waarde in het DL-register te zetten en de INT-instructie weer uit te voeren (vergeet niet IP weer op 100h te zetten.)

U hebt u misschien afgevraagd wat er gebeurd zou zijn als u had geprobeerd de traceer-opdracht bij de INT-instructie uit te voeren. We zullen doen alsof we de opdracht G 102 niet hebben gegeven en in plaats daarvan een stukje teruggaan om te zien wat er gebeurt. Als u dit zelf probeert, ga dan niet te ver: uw IBM-PC zou rare dingen kunnen gaan doen. Als u een paar stappen terug hebt getraceerd, verlaat Debug dan met de Q-opdracht. Die ruimt eventuele rommel op die u hebt achtergelaten.

-R

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD21          INT     21
```

-T

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0180 NV UP DI PL NZ NA PO NC
3372:0180 80FC4B          CMP     AH,4B
```

-T

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0183 NV UP DI NG NZ AC PE CY
3372:0183 7405          JZ      018A
```

-T

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0185 NV UP DI NG NZ AC PE CY
3372:0185 2E          CS:
3372:0186 FF2EAB0B      JMP     FAR [0BAB]          CS:0BAB=0BFF
```

-Q

Ziet u dat het eerste getal van het adres hier is veranderd, van 3970 in 3372? Deze laatste drie instructies waren een onderdeel van DOS, en het programma voor DOS zit in een ander segment. Er zijn zo nog veel meer instructies die DOS uitvoert voor het één enkel teken afdruckt; zelfs zo'n ogenschijnlijk eenvoudige taak is niet zo gemakkelijk als je zou denken. Nu ziet u ook waarom we de G-opdracht hebben gebruikt om ons programma op geheugenplaats 102h te draaien. Anders zouden we een stroom van instructies van DOS hebben gezien. (Als u een andere versie van DOS gebruikt dan wij, kunnen de instructies die u ziet er wat anders uitzien.)

3.2 Een net einde — INT 20h

Weet u nog dat onze INT-instructie 21h was? Als we de 21h veranderen in 20h, hebben we in plaats daarvan INT 20h. INT 20h is ook een interrupt-instructie, en zegt DOS dat we met het programma willen stoppen, zodat DOS weer de volledige besturing kan overnemen. In ons geval zal INT 20h de besturing weer overdragen aan Debug omdat we onze programma's vanuit Debug uitvoeren in plaats van vanuit DOS. Voer de instructie CDh 20h in, vanaf geheugenplaats 100 en probeer dan het volgende (vergeet niet de INT 20h-instructie te controleren met de R-opdracht:

-G 102

Program terminated normally

-R

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20 INT 20

-G

Program terminated normally

-R

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20 INT 20

De opdracht G, zonder enig getal erachter, voert het hele programma uit (dat nu uit slechts één instructie bestaat omdat INT 20 een *stop*-instructie is), en keert dan terug naar het begin. IP is weer op 100h gezet, waar we begonnen waren. De registers in dit voorbeeld zijn alleen 0 omdat we Debug weer opnieuw hebben gestart.

We kunnen deze INT 20h-instructie aan het einde van een programma gebruiken om netjes terug te keren naar DOS (of naar Debug), dus laten we deze instructie eens samen met INT 21h in een tweeregelig programma zetten.

3.3 Een programma van twee regels — de stukjes bij elkaar leggen

Voer vanaf geheugenplaats 100h achtereenvolgens de twee instructies INT 21h, INT 20h in (CDh 21h CD 20h). (Van nu af zullen we programma steeds beginnen op 100h.)

Als we maar één instructie hadden, konden we die instructie bekijken met de R-

opdracht, maar nu hebben we twee instructies. Om ze te kunnen zien, kunnen we de U (*Unassemble*)-opdracht geven, die net als de List-opdracht van BASIC werkt:

```
-U 100
3970:0100 CD21      INT    21
3970:0102 CD20      INT    20
3970:0104 40        INC    AX
3970:0105 20566F    AND     [BP+6F],DL
3970:0108 6C        DB     6C
3970:0109 756D      JNZ     0178
3970:010B 65        DB     65
3970:010C 20696E    AND     [BX+DI+6E],CH
3970:010F 206472    AND     [SI+72],AH
3970:0112 69        DB     69
3970:0113 7665      JBE     017A
3970:0115 2025      AND     [DI],AH
3970:0117 63        DB     63
3970:0118 2025      AND     [DI],AH
3970:011A 7300      JNB     011C
3970:011C D537      AAD     37
3970:011E 44        INC     SP
3970:011F 40        INC     AX
```

In de eerste twee instructies herkennen we de instructie die we net hebben ingevoerd. De andere instructies zijn resten die nog in het geheugen zaten. Naarmate ons programma langer wordt, zullen we meer eigen code in deze afbeelding zetten. Zet nu 02h in het AH-register en in het DL-register het nummer van een teken (net als u eerder deed toen u de registers AX en DX veranderde), tik dan gewoon de G-opdracht en uw teken verschijnt op het scherm. Als u bijvoorbeeld 41h in DL zet, ziet u dit:

```
-G
A
Program terminated normally
-
```

Probeer dit een paar keer voor we andere manieren laten zien om deze registers in te stellen.

3.4 Programma's invoeren

Van nu af zullen onze meeste programma's langer dan één instructie zijn, en om die programma's te laten zien, zullen we gebruik maken van een U(nassemble)-afbeelding. Ons laatste programma zou er dan zo uitzien.

```
3970:0100 CD21      INT    21
3970:0102 CD20      INT    20
```

Tot dusver hebben we de instructies voor onze programma's rechtstreeks als getallen, zoals CDh, 21h, ingevoerd. Maar dat is een hoop werk en, zoals blijkt, is er een veel eenvoudiger manier om instructies in te voeren.

Naast de U-opdracht kent Debug een A (*Assemble*)-opdracht, waarmee we de

mnemonische, door mensen leesbare, instructies rechtstreeks kunnen invoeren. Dus inplaats van cryptische getallen voor ons korte programma in te voeren, kunnen we de A-opdracht gebruiken om het volgende in te voeren:

```
-A 100
3970:0100 INT 21
3970:0102 INT 20
3970:104
-
```

Als u klaar bent uw instructies te assembleren, hoeft u alleen maar de Enter-toets in te drukken en verschijnt de Debug-prompt weer.

De A-opdracht vertelde Debug hier dat we onze instructies in mnemonische vorm wilden invoeren, en de 100 in onze opdracht betekende dat Debug het invoeren van de instructies moest beginnen op de geheugenplaats 100h. Omdat de assembleer-opdracht van Debug het invoeren van programma's veel eenvoudiger maakt, zullen we hem van nu af gebruiken om instructie in te voeren.

3.5 Gegevens in registers zetten

Hoewel we tot dusver erg veel met Debug hebben gedaan, zullen we er niet altijd programma's mee draaien. Normaal gesproken stelt een programma de registers AH en DL in vóór een INT 21h-instructie. Daartoe zullen we nog een instructie leren, genaamd MOV (van *move*, verplaatsen). Als we eenmaal voldoende van deze instructie weten, zullen we ons programmaatje voor het schrijven van een teken kunnen omzetten in een echt programma — een dat we rechtstreeks vanuit DOS kunnen uitvoeren. We zullen straks de MOV-instructie gebruiken om getallen in de registers AH en DL te zetten. Maar eerst leren we MOV gebruiken om getallen tussen registers te verplaatsen. Zet 1234h in AX (12h in het AH-register en 34h in AL) en ABCD in DX (ABh in DH en CDh in DL). Voer nu de volgende instructie in met de A-opdracht:

```
396F:0100 MOV AH,DL
```

Deze instructie *verplaatst* het getal in DL naar AH door een kopie ervan in AH te zetten; AL wordt er niet door veranderd. Als u deze ene regel traceert, zult u zien dat AX = CD34h en DX = ABCDh. Alleen AH is veranderd. Het bevat nu een kopie van het getal in DL.

Net als de BASIC-opdracht LET AH = DL, kopieert een MOV-opdracht een getal van het tweede register naar het eerste, en daarom zetten we AH vóór DL. Hoewel dit gebonden is aan enige beperkingen, waar we het verderop nog over zullen hebben, kunnen we andere vormen van de MOV-instructie gebruiken om getallen tussen andere registerparen te verplaatsen. Stel IP bijvoorbeeld opnieuw in en probeer dit eens:

```
396F:0100 MOV BX,AX
```

U hebt net woorden, in plaats van bytes, tussen registers verplaatst. De MOV-instructie werkt altijd tussen woorden en woorden, of bytes en bytes, nooit tussen

woorden en bytes. Dat is natuurlijk logisch, want hoe zou je een woord in een byte kunnen zetten?

We begonnen net met een getal in de registers AH en DL te zetten. Laten we dat nu eens doen met een andere vorm van de MOV-instructie:

```
396F:0100 MOV AH,02
```

Deze instructie zet 02h in het AH-register zonder dat dit van invloed is op het AL-register. De tweede byte van de instructie, 02h, is het getal dat we willen verplaatsen. Probeer eens een ander getal in AH te zetten: verander de tweede byte in een andere, bijvoorbeeld C1h, met de opdracht E 101.

We leggen nu alle stukjes van dit hoofdstuk bij elkaar en schrijven een langer programma — een dat helemaal zelfstandig een sterretje afdrukt, zonder dat we de registers (AH en DL) hoeven in te stellen. Het programma gebruikt MOV-instructies om AH en DL in te stellen voordat DOS met 21h wordt aangeroepen:

```
396F:0100 MOV AH,02
396F:0102 MOV DL,2A
396F:0104 INT 21
396F:0106 INT 20
```

Tik het programma in en controleer het met de U-opdracht (U 100). De listing ziet er dan zo uit:

396F:0100 B402	MOV	AH,02
396F:0102 B22A	MOV	DL,2A
396F:0104 CD21	INT	21
396F:0106 CD20	INT	20

Zorg dat IP naar geheugenplaats 100h wijst, probeer dan de G-opdracht om het hele programma te draaien. Op uw scherm moet u dan het sterretje zien:

```
-G
*
Program terminated normally
-
```

Laten we het volledige, zelfstandige programma dat we nu hebben, bewaren als een .COM-programma, zodat we het rechtstreeks vanuit DOS kunnen uitvoeren. We kunnen een .COM-programma vanuit DOS draaien door gewoon de naam ervan te tikken. Omdat ons programma nog geen naam heeft, moeten we het er een geven. De Debug-opdracht N (van *Name*, naam) geeft een bestand een naam voor we het naar de schijf schrijven. Tik:

```
-N SCHRSTER.COM
```

om ons programma de naam SCHRSTER.COM te geven. Deze opdracht schrijft ons bestand echter niet naar de schijf — hij geeft het alleen een naam.

Vervolgens moeten we Debug het aantal bytes opgeven dat ons programma bevat, zodat hij weet hoeveel geheugen we naar ons bestand willen schrijven. Als u de U-

listing van ons programma nog eens bekijkt, ziet u dat elke instructie twee bytes lang is (dat is niet altijd zo). We hebben vier instructies, dus ons programma is $4 * 2 = 8$ bytes lang. (We zouden ook de Debug-opdracht H voor ons kunnen laten werken om het aantal bytes in ons programma vast te stellen. Als u *H 108 100* tikt, waarbij 108 het adres is van de instructie na INT 20, krijgt u als uitkomst 8.)

Hebben we eenmaal het aantal bytes, dan moeten we dat ergens kwijt. Debug gebruikt het registerpaar BX: CX voor de lengte van ons bestand, dus door 8h in CX te zetten, vertellen we Debug dat ons programma acht bytes lang is. En omdat ons bestand maar acht bytes lang is, moeten we BX nog op nul zetten.

Omdat we de naam en de lengte van ons programma hebben aangegeven, kunnen we het nu met de Debug-opdracht W (van *Write*, schrijven) naar schijf schrijven:

```
-W
Writing 0008 bytes
-
```

We hebben nu een programma op onze schijf met de naam SCHRSTER.COM, dus laten we uit Debug stappen, met een Q, en het programma opzoeken. Gebruik de DOS-opdracht Dir om het bestand te *listen*.

```
C>DIR SCHRSTER.COM

Volume in drive C is FIXEDISK
Directory of C:>

SCHRSTER COM 8 29-05-88 13:52
 1 File(s) 99328 bytes free

C>
```

De directory laat zien dat SCHRSTER.COM op de schijf staat en dat hij acht bytes lang is, precies zoals het hoort. Om het programma te draaien, tikt u gewoon *schrster* achter de DOS-prompt. U ziet dat een sterretje op het scherm verschijnt. Niks aan.

3.6 Een reeks tekens schrijven

Als laatste voorbeeld van dit hoofdstuk gaan we INT 21h gebruiken met een ander functienummer in het AH-register, om een hele reeks tekens, een tekenstring, te schrijven. We zullen onze tekenstring in het geheugen moeten opslaan en DOS vertellen waar het die string kan vinden, dus intussen leren we ook meer over adressen en geheugen.

We hebben al gezien dat functienummer 02h bij INT 21h een teken op het scherm zet. Een andere functie, nummer 09h, drukt een hele string af en stopt daarmee wanneer ze een \$-teken in de string tegenkomt. We zetten eerst een string in het geheugen. We beginnen op geheugenplaats 200h, zodat de string niet verward raakt met de code voor ons programma. Voer de volgende getallen in met gebruik van de instructie E 200:

48	61	6C	6C
6F	2C	20	64
69	74	20	69
73	20	44	4F
53	2E	24	

Het laatste getal, 24h, is de ASCII-code voor een \$-teken, dat DOS vertelt dat dit het einde van onze tekenreeks is. U ziet zo wat deze string betekent, wanneer u het programma draait dat we nu invoeren:

```
396F:0100 MOV AH,09
396F:0102 MOV DX,0200
396F:0105 INT 21
396F:0107 INT 20
```

200h is het adres van de string die we hebben ingevoerd, en door 200h in het DX-register te zetten, vertellen we DOS waar het de string kan vinden. Controleer uw programma met de U-opdracht, draai het dan met de G-opdracht:

```
-G
Hallo, dit is DOS.
Program terminated normally
-
```

Nu we een aantal tekens in het programma hebben ingevoerd, is het tijd om nog een Debug-opdracht te leren kennen: D (van *Dump*). Deze opdracht 'dump't het geheugen naar het scherm, ongeveer zoals de U-opdracht instructies afbeeldt. Net als wanneer u de U-opdracht geeft, zet u gewoon een adres na de D om Debug te vertellen waar hij met de dump moet beginnen. Tik bijvoorbeeld de opdracht D 200 om een dump te zien van de reeks tekens die u net hebt ingevoerd:

```
-D 200
396F:0200 48 61 6C 6C 6F 2C 20 64-69 74 20 69 73 20 44 4F Hallo, dit is DO
396F:0210 53 2E 24 49 6E 76 61 6C-69 64 20 64 65 76 69 63 S.$Invalid devic
.
.
.
```

Na elk adressenpaar (zoals 396F:0200 in ons voorbeeld) zien we 16 hex-bytes, gevolgd door de 16 ASCII-tekens voor deze bytes. Zo ziet u op de eerste regel de meeste ASCII-codes en tekens die u hebt ingetikt. Het afsluitende \$-teken dat u hebt getikt, is het derde teken op de tweede regel; de rest van die regel is een mengelmoes van allerlei tekens.

Een punt in het ASCII-venster stelt een echte punt of een speciaal teken voor, zoals de Griekse letter pi. De Debug-opdracht D toont maar 96 van de 256 tekens van de IBM-PC-tekenset, dus voor de overige 160 tekens wordt een punt gebruikt.

We zullen de D-opdracht in de toekomst gebruiken om getallen te controleren die we als gegevens hebben ingevoerd, of die getallen nu tekens zijn of uit gewone cijfers bestaan. (Zie voor nadere informatie het deel over Debug in uw DOS-handleiding.) Ons programma voor het afdrukken van de string is nu klaar, dus we kunnen het

naar de schijf schrijven. Dat gaat net zo als bij het schrijven van SCHRSTER.COM naar de schijf, behalve dat we deze keer onze programmalengte moeten instellen op een waarde die groot genoeg is om de string op 200h te kunnen bevatten. Ons programma begint op regel 100h, en aan de geheugendump die we net hebben uitgevoerd, zien we dat het eerste teken (.) na het \$-teken die onze string afsluit op 212h staat. Weer kunnen we de H-opdracht gebruiken om het verschil tussen deze twee getallen te bepalen. Kijk hoeveel $212h - 100h$ is en sla deze waarde op in het CX-register en zet BX weer op nul. Gebruik de N-opdracht om het programma een naam te geven (zet de .COM-uitbreiding erachter om het programma rechtstreeks vanuit DOS te kunnen draaien), geef dan de W-opdracht om het programma en de gegevens naar een schijfbestand te schrijven.

Dat was alles wat het schrijven van tekens naar het scherm betreft — afgezien van nog een laatste opmerking: u hebt misschien opgemerkt dat DOS het \$-teken helemaal niet verstuurt. Heel juist, want DOS gebruikt het \$-teken om het einde van een tekenstring te markeren. Dat betekent dat we DOS niet kunnen gebruiken om een string met een \$ erin af te drukken, maar in een hoofdstuk verderop zullen we zien hoe een string met een \$-teken of elk ander speciaal teken moet worden afgebeeld.

3.7 Samenvatting

Na onze voorbereidingen in de eerste twee hoofdstukken waren we zover dat we aan een echt programma konden gaan werken. In dit hoofdstuk hebben we onze kennis van hex-getallen, Debug, 8088-instructies en het geheugen gebruikt om een kort programmaatje te schrijven dat een teken en een tekenstring op het scherm afdruckte. Daarbij leren we ook enkele nieuwe dingen.

Eerst leerden we over INT-instructies — niet erg gedetailleerd, maar voldoende om twee kleine programmaatjes te kunnen schrijven. In volgende hoofdstukken komen we meer te weten over interrupt-instructies wanneer we meer inzicht krijgen in de 8088-microprocessor die onder de kap van uw IBM-PC schuilgaat.

Debug is weer eens een nuttige en trouwe gids gebleken. We hebben veel gebruik gemaakt van Debug bij het afbeelden van de inhoud van registers en geheugen, en in dit hoofdstuk hebben we er nog meer mogelijkheden van leren toepassen. Debug heeft ons programmaatje gedraaid met de G-opdracht.

We hebben ook geleerd over de stop-instructie INT 20, en de MOV-instructie waarmee getallen naar en tussen registers kunnen worden verplaatst. Met de stop-instructie (INT 20) konden we een volledig programma bouwen dat we naar de schijf konden schrijven en rechtstreeks vanuit DOS draaien zonder de hulp van Debug. En met de MOV-instructie konden we voorafgaande aan een INT 21 (afdruk)-instructie registers instellen zodat we een zelfstandig programma konden schrijven waarmee een teken op het scherm wordt afgedrukt.

Ten slotte hebben we het hoofdstuk afgerond met de functie INT 21h, waarmee een hele tekenstring kan worden afgebeeld. We zullen al deze instructies verderop in dit boek nog veel gebruiken, maar zoals u bij het gebruik van de A- en de U-opdracht van Debug wel zult hebben gezien, hoeft u de machinecode voor deze instructies niet te onthouden.

We weten nu voldoende om door te gaan naar het afdrukken van binaire getallen. In het volgende hoofdstuk schrijven we een programmaatje dat een byte op het scherm afdruckt als een reeks binaire cijfers (enen en nullen).

4 Binaire getallen afdrukken

- 4.1 Rotaties en de overdrachtvlag 58**
- 4.2 Optellen met de overdrachtvlag 59**
- 4.3 Lussen 60**
- 4.4 Een binair getal schrijven 62**
- 4.5 De Proceed-opdracht 63**
- 4.6 Samenvatting 64**

In dit hoofdstuk gaan we een programma schrijven waarmee binaire getallen als reeksen enen en nullen op het scherm worden afgedrukt. Over het merendeel van de benodigde kennis beschikken we al, en ons werk hier zal u de reeds behandelde stof nog beter helpen begrijpen. We zullen ook nog enkele instructies toevoegen aan die welke we al kennen, met inbegrip van een andere versie van ADD en opdrachten waarmee we delen van ons programma kunnen herhalen. Maar eerst gaan we iets geheel nieuws leren.

4.1 Rotaties en de overdrachtvlag

In hoofdstuk 2, toen we voor het eerst kennismakten met het rekenen in hex, zagen we dat het optellen van 1 en FFFFh een som 10000h moest geven, maar dat gebeurde niet. Slechts de vier hex-cijfers aan de rechterkant passen in een woord: de 1 kan er niet meer bij. We zagen ook dat deze 1 een overloop is en niet verloren gaat. Waar gaat hij heen? Hij wordt gezet in wat een *vlag* heet — in dit geval de overdrachtvlag, ofwel CF (*Carry Flag*). Vlaggen bevatten getallen van één cijfer, en kunnen dus óf een 1 óf een 0 bevatten. Als we een 1 naar het vijfde hex-cijfer willen overdragen, gaat hij naar de overdrachtvlag.

We gaan even terug naar onze ADD-instructie van hoofdstuk 2 (ADD AX,BX). Zet FFFFh in AX en 1 in BX, traceer dan de ADD-instructie. Aan het eind van de tweede regel van de R-afbeelding van Debug ziet u acht letterparen. Het laatste daarvan, dat NC of CY kan zijn, is de overdrachtvlag. Omdat uw optel-instructie tot een overloop van 1 heeft geleid, zult u nu zien dat de overdrachtstatus CY (*Carry Yes*) is. Het overdracht-bit is nu 1 of, zoals we dat zullen uitdrukken, het is *gezet*.

Alleen om te bevestigen dat we hier een zeventiende bit hebben opgeslagen (bij een optelling van bytes zou het het negende bit zijn), moet u eens 1 optellen bij de 0 in AX door IP weer op 100 te zetten en de optel-instructie weer te traceren. De overdrachtvlag wordt beïnvloed door elke optel-instructie, en deze keer dient er geen sprake van overloop te zijn, dus de overdrachtvlag moet op 0 gezet zijn. En de overdracht is inderdaad nul, zoals de NC (*No Carry*) in de register-afbeelding ook aangeeft.

(Verderop leren we meer over de andere statusvlaggen, maar als u het nu al graag wilt weten: u kunt er informatie over vinden onder de R-opdracht van Debug in uw DOS-handleiding.)

Nu gaan we eens zien hoe het afdrukken van een binair getal in z'n werk gaat, om te kijken of de informatie over de overdrachtvlag van nut kan zijn. We drukken maar één teken tegelijk op het scherm af, en willen de bits één voor één, vanaf de linkerkant, van ons getal halen. Het eerste teken dat we bijvoorbeeld van het getal 1000 0000b zouden willen afdrukken, is de 1. Als we die hele byte nu één plaats naar links zouden kunnen verschuiven, de 1 in de overdrachtvlag laten vallen en er aan de rechterkant een 0 bijzetten, en dit proces dan bij elk volgende cijfer herhalen, zou de overdrachtvlag steeds onze binaire cijfers er afhalen. En dat is mogelijk met een nieuwe instructie genaamd RCL (*Rotate Carry Left*).

Om te zien hoe dit werkt, moet u het volgende programmaatje invoeren:

```
3985:0100 D0D3          RCL     BL,1
```


Deze instructie *roteert* de byte in BL één bit naar links (vandaar de ,1) en doet dat door middel van de overdrachtvlag. Deze instructie wordt roteerinstructie genoemd omdat RCL het meest linkse bit in de overdrachtvlag zet, en het bit dat op dat moment in de overdrachtvlag staat op de meest rechtse bit-positie (0) zet. Daarbij worden alle andere bits naar links verplaatst, of geroteerd. Na voldoende rotaties (zeventien voor een woord, negen voor een byte) komen de bits weer op hun oorspronkelijke plaats te staan, en hebt u het oorspronkelijke getal weer.

Zet B7h in het BX register, traceer deze roteer-instructie meerdere malen. In binair omgezet zien de resultaten er als volgt uit:

<u>Overdrachtvlag</u>	<u>BL-register</u>		
0	1 0 1 1 0 1 1 1	BFh	hier beginnen we
1	0 1 1 0 1 1 1 0	6Eh	
0	1 1 0 1 1 1 0 1	DDh	
1	1 0 1 1 1 0 1 0	BAh	
	.		
0	1 0 1 1 0 1 1 1	B7h	na 9 rotaties

Bij de eerste rotatie schuift bit 7 van BL naar de overdrachtvlag, schuift het bit in de overdrachtvlag naar bit 0 van BL, en schuiven alle andere bits een positie naar links. Bij opeenvolgende verschuivingen worden de bits steeds naar links geroteerd tot, na negen rotaties, het oorspronkelijke getal weer in het BL-register staat.

We zijn weer een stap dichterbij het maken van ons programma voor het schrijven van binaire getallen gekomen, maar er ontbreken nog een paar stukjes. Laten we eerst eens bekijken hoe we het bit in de overdrachtvlag in het teken 0 of 1 kunnen omzetten.

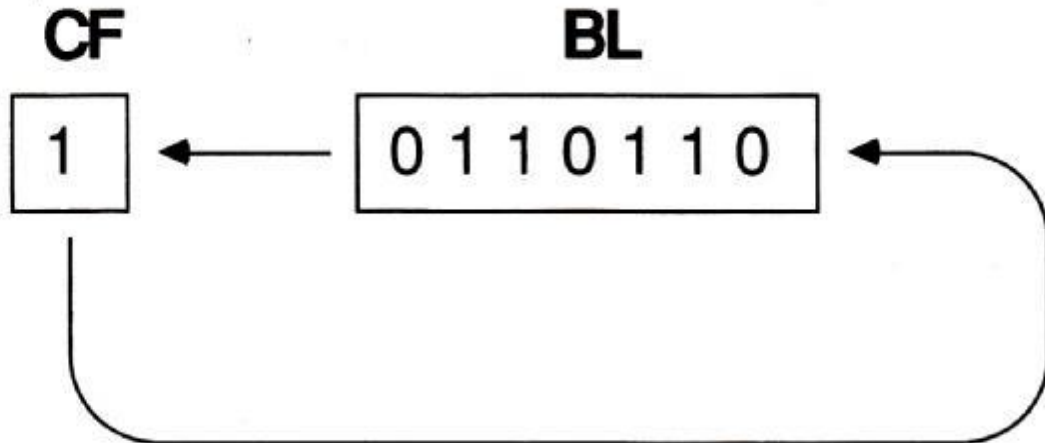
4.2 Optellen met de overdrachtvlag

Bij de gewone optel-instructie, bijvoorbeeld ADD AX,BX, worden gewoon twee getallen opgeteld. Een andere instructie, ADC (*Add with Carry*) telt drie getallen bij elkaar op: de twee bekende en nog een bit van de overdrachtvlag. Als u uw ASCII-tabel naslaat, zult u zien dat 30h het teken 0 is, en 31h het teken 1. Optellen van de overdrachtvlag en 30h geeft dus het teken 0 als de vlag leeg is, en 1 als de vlag gezet is. Als DL = 0 en de overdrachtvlag is gezet (1), heeft uitvoering van

```
ADC DL,30
```

tot resultaat dat DL (0) wordt opgeteld bij 30h ('0') en bij 1h (de overdracht) met als uitkomst 31h ('1'). En met één instructie hebben we de overdracht omgezet in een teken dat we kunnen afbeelden.

In plaats van nu een voorbeeld van ADC door te nemen, wachten we tot ons programma helemaal klaar is. We nemen dan de instructies één voor één door, volgens een procedure die *single-stepping* (stap voor stap) wordt genoemd. We zullen daarbij zien hoe de ADC-instructie werkt en hoe mooi ze in het programma past. Maar eerst



Afb. 4-1. De instructie *RCL BL,1*.

moeten we nog een andere instructie leren, die we zullen gebruiken om onze instructies RCL, ADC en INT 21H (afdrukken) acht keer te herhalen; een keer voor elk bit in een byte.

4.3 Lussen

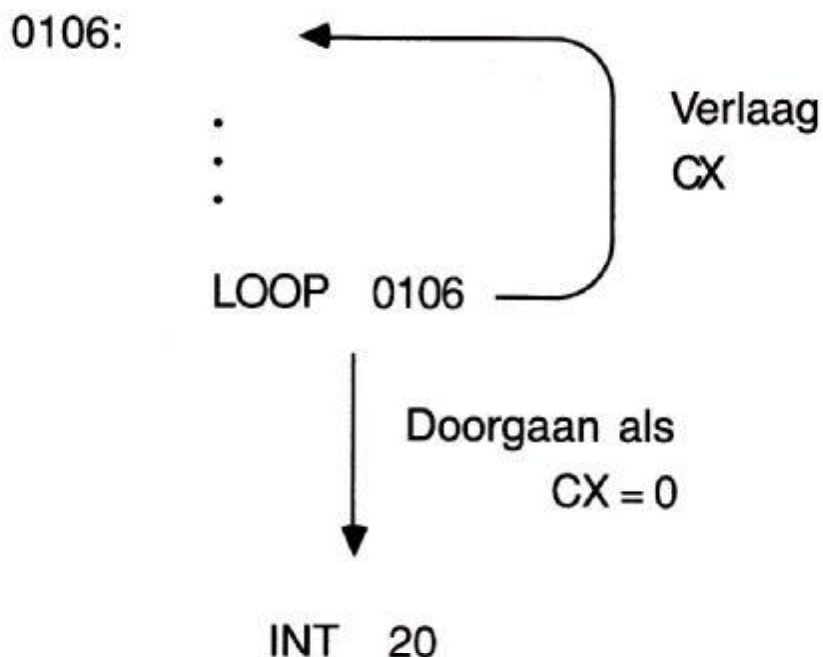
Zoals gezegd, beperkt de RCL-instructie zich niet alleen tot het roteren van bytes; ze kan hele woorden roteren. We zullen deze eigenschap gebruiken om de instructie *LOOP* te demonstreren. *LOOP* heeft iets weg van de *FOR-NEXT*-lus in BASIC, maar is niet zo algemeen. Net als de *FOR-NEXT*-lus van BASIC moeten we *LOOP* vertellen hoe vaak de lus moet worden doorlopen. We doen dat door ons herhalingsgetal in het CX-register te zetten. Bij elke slag door de lus trekt de 8088 een van CX af, en als CX gelijk aan nul wordt, beëindigt *LOOP* de lus.

Waarom het CX-register? De C in CX staat voor *Count* (tellen). We kunnen dit register als een register voor algemene doeleinden gebruiken, maar zoals u in het volgende hoofdstuk zult zien, wordt het CX-register ook samen met andere registers gebruikt wanneer we bewerkingen willen herhalen.

Hier is een eenvoudig programmaatje dat het BX-register acht keer roteert, en BL in BH zet (maar niet omgekeerd, omdat we via de overdrachtvlag roteren):

396F:0100	BBC5A3	MOV	BX,A3C5
396F:0103	B90800	MOV	CX,0008
396F:0106	D1D3	RCL	BX,1
396F:0108	E2FC	LOOP	0106
396F:010A	CD20	INT	20

Onze lus begint op 106h (RCL BX,1) en eindigt met de *LOOP*-instructie. Het getal na *LOOP* (106h) is het adres van de RCL-instructie. Als we het programma draaien, trekt *LOOP* 1 van CX af en springt naar adres 106h als CX nog geen 0 is. De instructie RCL BX,1 (*Rotate Carry Left*, roteer links, met overdracht) wordt hier acht keer uitgevoerd omdat CX vóór de lus op 8 is gezet.



Afb. 4-2. De LOOP-instructie.

Het is u misschien opgevallen dat de LOOP-instructie, anders dan de FOR-NEXT-lus in BASIC, aan het einde van de lus staat (waar we in BASIC de NEXT-opdracht zetten). En het begin van de lus, de RCL-instructie op 106h, heeft geen speciale instructie zoals FOR in BASIC. Als u een taal als Pascal kent, ziet u dat de LOOP-instructie enigszins verwant is met de instructie REPEAT-UNTIL in die taal, waarbij de REPEAT-instructie gewoon het begin van de te doorlopen groep instructies aangeeft. U kunt ons programmaatje op verschillende manieren uitvoeren. Als u gewoon G tikt, zult u geen veranderingen in de register-afbeelding waarnemen, omdat Debug alle registers bewaart voordat hij een G-opdracht uitvoert. Als hij daarna een INT 20-instructie tegenkomt (zoals in ons programma) herstelt hij alle registers weer. Probeer G maar eens. U zult zien dat IP weer op 100h is gezet (waar u begon), en dat de andere registers er ook niet anders uitzien.

Als u het geduld ervoor hebt, kunt u dit programma ook traceren. Door een stap tegelijk te zetten, kunt u zien hoe bij elke stap de registers veranderen:

```

-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
OCDE:0100 BBC5A3      MOV     BX,A3C5
-T

AX=0000 BX=A3C5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0103 NV UP DI PL NZ NA PO NC
OCDE:0103 B90800      MOV     CX,0008
-T

AX=0000 BX=A3C5 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 NV UP DI PL NZ NA PO NC
OCDE:0106 D1D3        RCL     BX,1
-T
  
```



```

AX=0000 BX=478A CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 OV UP DI PL NZ NA PO CY
OCDE:0108 E2FC          LOOP    0106
-T

```

```

AX=0000 BX=478A CX=0007 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 OV UP DI PL NZ NA PO CY
OCDE:0106 D1D3          RCL     BX,1
.
.
.
-T

```

```

AX=0000 BX=C551 CX=0001 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 NV UP DI PL NZ NA PO CY
OCDE:0108 E2FC          LOOP    0106
-T

```

```

AX=0000 BX=C551 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=010A NV UP DI PL NZ NA PO CY
OCDE:010A CD20          INT     20
-T

```

U kunt in plaats hiervan ook G 10A tikken om het programma tot aan, maar niet tot en met, de INT 20-instructie op 10Ah uit te voeren; de registers zullen dan het resultaat van het programma laten zien.

Als u dit probeert, zult u zien dat CX = 0 en BX = C551 of

BX = C5D1, afhankelijk van de waarde van de overdrachtvlag voordat u het programma draaide. De C5 die de MOV-instructie van ons programma bij het begin in BL zette, staat nu in het BH-register, maar BL bevat niet A3 omdat we BX *door* de overdrachtvlag hebben geroteerd. Later zullen we andere manieren van roteren bekijken zonder de overdrachtvlag te doorlopen. Eerst gaan we terug naar ons doel om een getal in de binaire notatie af te drukken.

4.4 Een binair getal schrijven

We hebben gezien hoe we binaire cijfers één voor één kunnen weghalen, en in ASCII-tekens omzetten. Als we er voor het afdrukken van onze cijfers een INT 21 aan toevoegen, is ons programma klaar. Hier is het programma; de eerste instructie zet AH op 02 voor de functieaanroep van INT 21h (02 vertelt DOS immers dat het het teken in het DL-register moet afdrukken).

```

3985:0100 B402          MOV     AH,02
3985:0102 B90800        MOV     CX,0008
3985:0105 B200          MOV     DL,00
3985:0107 D0D3          RCL     BL,1
3985:0109 80D230        ADC     DL,30
3985:010C CD21          INT     21
3985:010E E2F5          LOOP    0105
3985:0110 CD20          INT     20

```


We hebben gezien hoe alle stukjes werken, en kunnen ze nu bij elkaar leggen. Gebruik roteer BL (met de instructie RCL BL,1) om de bits er één voor één af te halen, kies een getal dat u in binair afgedrukt wilt zien, laad dat in het BL-register, en draai dan dit programma met een G-opdracht. Na de INT 20h-instructie geeft de G-opdracht de registers weer de waarden die ze eerst hadden, dus BL bevat dan nog het getal dat u in binair hebt afgedrukt.

De instructie ADC DL,30 in ons programma zet de overdrachtvlag om in het teken nul of een. De instructie MOV DL,0 maakt DL eerst nul, daarna telt de ADC-instructie 30h bij DL op, en ten slotte de overdrachtvlag daarbij op. Omdat 30h de ASCII-code voor een 0 is, geeft ADC DL,30 een 0 wanneer de overdrachtvlag leeg is (NC) of 1 wanneer de vlag gezet is (CY).

Als u wilt zien wat er gebeurt wanneer u dit programma draait, traceer het dan helemaal. Onthoud dat u bij het stap voor stap doornemen met de T-opdracht een beetje voorzichtig moet zijn. Hij bevat namelijk een INT 21h-instructie en, zoals u zag toen we INT 21h de eerste keer tegenkwamen, doet DOS voor die ene instructie een hele hoop werk. Daarom kunt u T niet bij de INT 21 gebruiken.

U kunt echter alle andere instructies in dit programma echter wel traceren, behalve de laatste INT 20, die pas helemaal aan het eind van belang is. Tik iedere keer de opdracht G 10E als u tijdens het traceren de lus doorloopt en de INT 21h-instructie tegenkomt. Uw G-opdracht, gevolgd door een adres, vertelt Debug dat hij het programma moet blijven draaien, maar stoppen wanneer IP gelijk wordt aan het adres (10E) dat u hebt opgegeven. Dat wil zeggen, Debug zal de instructie INT 21h uitvoeren zonder dat u er doorheen traceert, maar stoppen voor uitvoering van de LOOP-instructie op 10E, zodat u weer kunt beginnen met het programma te traceren. Het getal dat u na G tikt, wordt in de DOS-handleiding een *breakpoint* genoemd; breakpoints zijn erg nuttig wanneer u de interne werking van programma's probeert te begrijpen.

Beëindig ten slotte het programma wanneer u bij de INT 20h-instructie aankomt met de G-opdracht zonder meer.

4.5 De Proceed-opdracht

Of u nu al of niet hebt geprobeerd de instructies in het programma te traceren, u hebt in elk geval gezien dat we met een instructie als G 10E over een INT-instructie heen kunnen traceren die, zeg, op 10C begint. Maar dat betekent dat we iedere keer als we over een INT-instructie heen traceren, het adres moeten opzoeken van de instructie die meteen ná de INT-instructie staat.

Er blijkt echter een Debug-opdracht te bestaan die het traceren voorbij INT-instructies veel eenvoudiger maakt. De P-opdracht (van *Proceed*, doorgaan) doet al het werk voor ons. Om te zien hoe, moet u het programma eens traceren, maar dan iedere keer als u de INT 21h-instructie bereikt P tikken, inplaats van G 10E, zoals we net zeiden.

We zullen in de rest van dit boek veel gebruik maken van de P-opdracht omdat het een prettige manier is om te traceren voorbij opdrachten als INT, die grotere programma's aanroepen zoals de routines van DOS. Voor we echter verder gaan, moeten we een ding over de P-opdracht zeggen — hij wordt niet vermeld in de versies van DOS vóór 3.00. Misschien is men vergeten deze opdracht te beschrijven of, wat waar-

schijnlijker is, Microsoft had geen tijd meer om de P-opdracht volledig te testen voor het uitbrengen van de DOS-versie 2.00. Wat de reden ook moge zijn, als u een DOS-versie vóór 3.00 hebt, moet u beseffen dat de P-opdracht *misschien niet* altijd goed werkt — hoewel we er zelf nooit problemen mee hebben gehad als we hem gebruikten.

Dat was zo'n beetje alles wat we moeten doen voor het afdrukken van getallen als reeksen enen en nullen, maar we geven u nog een eenvoudige oefening op. Probeer eens dit programma zo te wijzigen dat het na uw binaire getal ook nog een *b* afdruckt (aanwijzing: de ASCII-code voor *b* is 62h).

4.6 Samenvatting

In dit hoofdstuk hadden we de gelegenheid om even op adem te komen na al ons zwoegen bij de nieuwe onderwerpen die we in de hoofdstukken 1 tot en met 3 hadden behandeld. Dus waar zijn we geweest en wat hebben we gezien?

We zijn voor het eerst vlaggen tegengekomen, en hebben de overdrachtvlag bekeken, die hier van speciale betekenis was omdat hij het heel gemakkelijk voor ons maakte om een binair getal af te drukken. Hij deed dat zodra we de roteer-instructie RCL hadden leren kennen, die een byte of woord met één bit tegelijk naar links verschuift. Toen we over de overdrachtvlag en het roteren van bytes en woorden hadden geleerd, pakten we een nieuwe versie van de optel-instructie ADC op en waren bijna klaar om ons programma voor het afdrukken van een getal in de binaire notatie te gaan opzetten.

Hier verscheen de LOOP-instructie ten tonele. Door een lus-teller in het CX-register te zetten, konden we de 8088-instructies een aantal keren in een lus laten uitvoeren. We zetten CX op 8, om een lus acht keer te herhalen.

Dat is alles wat we nodig hadden om ons programma te schrijven. We zullen dit gereedschap in de volgende hoofdstukken weer gebruiken. In het hoofdstuk hierna gaan we een binair getal in de hexadecimale notatie afbeelden, net zoals Debug dat doet, dus als we klaar zijn met hoofdstuk 5, hebben we ook een beter idee van hoe Debug getallen van binair in hex vertaalt. Daarna gaan we naar het andere eind van Debug: getallen lezen die in hex worden ingetikt en ze dan omzetten in de binaire notatie van de 8088.

5 Afdrukken in hex

- 5.1 Vergelijk- en statusbits 66**
- 5 Een enkel hex-cijfer afdrukken 68**
- 5.3 Nog een roteer-instructie 71**
- 5.4 Logica en AND 72**
- 5.5 Alles bij elkaar leggen 73**
- 5.6 Samenvatting 74**

Ons programma van hoofdstuk 4 was vrij rechttoe, rechtaan. We hadden geluk omdat de overdrachtvlag het gemakkelijk maakte een binair getal als een reeks nullen en enen af te beelden. We gaan nu door naar het afdrukken van getallen in hex. We zullen dit wat minder rechtstreeks moeten aanpakken, en zullen onszelf herhalen in onze programma's en dezelfde reeks instructies meermalen neerschrijven. Maar dat soort herhaling duurt niet eeuwig: in hoofdstuk 7 leren we meer over procedures (subprogramma's) die de noodzaak om een zelfde reeks instructies meer dan één keer te schrijven, wegnemen. Laten we eerst nog wat nuttige instructies leren en kijken hoe getallen in hex moeten worden afgedrukt.

5.1 Vergelijk- en statusbits

In het vorige hoofdstuk leerden we iets over statusvlaggen en onderzochten de overdrachtvlag, die in de register-afbeelding van Debug als CY of NC wordt voorgesteld. De andere vlaggen, die even nuttig zijn, houden de *status* van de laatste rekenkundige bewerking bij. In totaal zijn er acht vlaggen, en we zullen die bekijken zodra er aanleiding toe is.

U zult nog weten dat CY betekent dat de overdrachtvlag 1, of gezet, is, terwijl NC betekent dat de overdrachtvlag 0 is. Bij alle vlaggen betekent 1 *waar* en 0 *onwaar*. Als u bijvoorbeeld een SUB-instructie uitvoert met als uitkomst 0, wordt de vlag die bekend staat als de nulvlag op 1 — waar — gezet en zou u dat in de register-afbeelding als ZR (Zero) zien. Anders zou de nulvlag weer op 0 — NZ (*Not Zero*) — worden gezet.

Laten we eens een voorbeeld bekijken waarin de nulvlag wordt getest. We gebruiken daarvoor de SUB-instructie om twee getallen van elkaar af te trekken. Als de twee getallen gelijk zijn, zal de uitkomst nul zijn, en zal de nulvlag als ZR in uw afbeelding verschijnen. Voer de volgende aftrek-instructie in:

```
396F:0100 29D8          SUB     AX,BX
```

Traceer nu de instructie met verschillende getallen, en kijk of u ZR of NZ in de nulvlag ziet. Als u in zowel het AX- als het BX-register hetzelfde getal zet (zoals F5h in het volgende voorbeeld) zult u zien dat de nulvlag na een aftrek-instructie wordt gezet, en na de volgende weer wordt gewist:

```
-R
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
OCDE:0100 29D8          SUB     AX,BX
-T
```

```
AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI PL ZR NA PE NC
OCDE:0102 37          AAA
-R IP
IP 0102
:100
```


-R

AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL ZR NA PE NC
OCDE:0100 29D8 SUB AX,BX

-T

AX=FF0B BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI NG NZ AC PO CY
OCDE:0102 37 AAA

Als we nu 1 van 0 aftrekken, is de uitkomst FFFFh, wat, zoals we in hoofdstuk 1 hebben gezien, 1 in de twee-complementvorm is. Kunnen we nu aan de registerafbeelding zien of een getal positief of negatief is? Ja, een andere vlag, de *tekenvlag* (Sign flag), wisselt tussen NG (*Negatief*) en PL (*Plus*) en wordt op 1 gezet wanneer een getal een negatief twee-complementgetal is.

Een andere nieuwe vlag die voor ons van belang is, is de overloopvlag, die wisselt tussen OV (*Overflow*, overloop) wanneer de vlag 1 is, en NV (*No Overflow*, geen overloop) wanneer de vlag 0 is. De overloopvlag wordt gezet als het tekenbit verandert wanneer dat niet zou moeten. Als we bijvoorbeeld twee positieve getallen optellen, zeg 7000h en 6000h, krijgen we een negatief getal D000h, ofwel -12288. Dat is fout omdat de uitkomst een overloop van het woord geeft. De uitkomst dient positief te zijn, maar is dat niet, zodat de 8088 de overloopvlag zet. (Denk eraan dat, als we met getallen zonder teken werken, dit geen fout zou zijn en we de overloopvlag zouden kunnen negeren.)

Probeer eens verschillende getallen uit om te zien of u elk van deze vlaggen kunt zetten en herstellen, en ga ermee door tot u er vertrouwd mee bent. Trek voor de overloop een groot negatief getal van een groot positief getal af — bijvoorbeeld 7000h - 8000h, omdat 8000h in de twee-complementvorm gelijk is aan -32768.

We zijn nu zover dat we een groep instructies kunnen gaan bekijken die bekend staat onder de naam *voorwaardelijke* spronginstructies. Ze bieden ons de kans om statusvlaggen op een gemakkelijker manier te controleren dan tot dusver. De instructie JZ (*Jump if Zero*, spring indien nul) springt naar een nieuw adres als de laatste rekenkundige uitkomst nul was. Als we dus een SUB-instructie door, zeg, JZ 15A laten volgen, zou een uitkomst nul van het verschil ertoe leiden dat de 8088 springt naar, en begint met het uitvoeren van, opdrachten op het adres 15Ah in plaats van met de volgende instructie.

De JZ-instructie test de nulvlag, en voert, als die gezet (ZR) is, een sprong uit net als bijvoorbeeld bij de BASIC-opdracht IF A = 0 THEN 100. Het tegenovergestelde van JZ is JNZ (*Jump if Not Zero*, spring indien niet nul). Laten we eens een eenvoudig voorbeeld nemen waarin JNZ wordt gebruikt en van een getal wordt afgetrokken tot de uitkomst nul is:

396F:0100 2C01 SUB AL,01
396F:0102 75FC JNZ 0100
396F:0104 CD20 INT 20

Zet bijvoorbeeld drie in AL, zodat u de lus een paar keer doorloopt, en neem dit programma met één instructie tegelijk door om te zien hoe de voorwaardelijke vertakkingen werken. We hebben de instructie INT 20h er aan het eind bijgezet om te

zorgen dat u door het tikken van G niet voorbij het einde van uw programma gaat: het is een goede voorzorgsmaatregel.

Het is u misschien opgevallen dat het gebruik van SUB om twee getallen te vergelijken, zoals we net hebben gedaan, tot het mogelijk ongewenste resultaat leidt dat het eerste getal gewijzigd wordt. Er is nog een andere instructie, *CMP* (*Compare*, vergelijk) waarmee we kunnen aftrekken zonder de uitkomst ergens op te slaan en zonder het eerste getal te veranderen. De uitkomst wordt alleen gebruikt om de vlaggen te zetten, dus we kunnen één van de vele voorwaardelijke spronginstructies gebruiken na een vergelijking. Om te zien wat er gebeurt, moet u eerst AX en BX dezelfde inhoud, F5h, geven en dan deze instructie traceren:

```
-A 100
```

```
OCDE:0100 CMP AX,BX
```

```
OCDE:0102
```

```
-T
```

```
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=OCDE ES=OCDE SS=OCDE CS=OCDE IP=0102 NV UP DI PL ZR NA PE NC
OCDE:0102 3F AAS
```

De nulvlag is nu gezet (ZR), maar in beide registers blijft F5h staan.

Nu gaan we CMP gebruiken om één enkel hex-cijfer af te drukken. Daartoe schrijven we een aantal instructies die vlaggen gebruiken om, zoals LOOP in het vorige hoofdstuk, de stroom van ons programma te wijzigen, op dezelfde manier als de IF-THEN-opdracht in BASIC. Deze nieuwe instructies gebruiken de vlaggen om te testen op condities als kleiner dan, groter dan enz. We hoeven ons daarbij niet te bekommeren over welke vlaggen gezet zijn wanneer het eerste getal kleiner is dan het tweede; de instructies weten naar welke vlaggen ze moeten kijken.

5.2 Een enkel hex-cijfer afdrukken

Laten we beginnen met een klein getal (tussen 0 en Fh) in het BL-register te zetten. Omdat elk getal tussen 0 en Fh gelijk is aan een hex-cijfer, kunnen we onze keuze omzetten in een enkel ASCII-teken en dat dan afdrukken. We gaan eens kijken welke stappen we moeten zetten om deze omzetting tot stand te brengen.

De ASCII-teken 0 tot en met 9 hebben de waarden 30h tot en met 39h; de tekens A tot en met F hebben echter de waarden 41h tot en met 46h. Dat geeft een probleem: deze twee groepen ASCII-teken worden door zeven tekens gescheiden. Als gevolg daarvan zal de omzetting naar ASCII voor de twee groepen getallen (0 tot en met 9 en Ah tot en met Fh) verschillen, dus we moeten elke groep op een verschillende manier aanpakken. Een BASIC-programma om deze tweeledige omzetting tot stand te brengen, zou er zo uitzien:

```
100 IF BL < &H0A
    THEN BL = BL + &H30
    ELSE BL = BL + &H37
```

(Merk op dat we 0Ah, en niet AH, voor het getal A hebben geschreven om het getal Ah niet te verwarren met het register AH. We zullen in dit soort situaties vaak een

Teken	ASCII Code (Hex)
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47

Afb. 5-1. Deel van ASCII-tabel met de tekens die door hex-cijfers worden gebruikt.

nul voor hex-getallen zetten, omdat er anders verwarring kan ontstaan. In feite doet u er verstandig aan voor *alle* hex-getallen een nul te zetten omdat het nooit kwaad kan.)

Ons omzettingsprogramma in BASIC is vrij eenvoudig. Helaas kent de machinetaal van de 8088 geen ELSE-opdracht; hij is veel primitiever dan BASIC, dus we zullen wat slimmer moeten zijn. We laten nog een BASIC-programma zien, nu een dat de methode volgt die we voor ons machinetaal-programma zullen gebruiken:

```
100 BL = BL + &H30
110 IF BL >= &H3A
    THEN BL = BL + &H7
```

U kunt zelf nagaan of dit programma werkt door het met geschikte voorbeelden te testen. De getallen 0, 9, Ah en Fh zijn daarvoor het meest geschikt omdat ze gevierende *grens*-condities bestrijken — de gebieden waar we vaak op moeilijkheden stuiten.

0 en Fh zijn hier respectievelijk het kleinste en het grootste hex-getal van één cijfer, dus met 0 en Fh als voorbeeld kunnen we de boven- en onderkant van ons bereik controleren. Hoewel ze naast elkaar staan, vereisen de getallen 9 en 0Ah twee verschillende omzettingsmethoden in ons programma. Door het gebruik van 9 en 0Ah kunnen we zien of we de juiste plaats hebben gekozen om van de ene omzettingmethode op de andere over te gaan.

De machinetaal-versie van dit programma bevat nog een paar stappen meer, maar is in wezen gelijk aan de BASIC-versie. Ze gebruikt zowel de CMP-instructie als een voorwaardelijk spronginstructie genaamd JL (*Jump if Less than*, spring indien kleiner). Dit is het programma waarmee een hex-getal van één cijfer uit het BL-register in hex wordt afgedrukt:

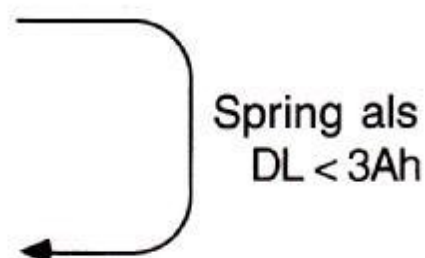
8510:0100 B402	MOV	AH,02
3985:0102 88DA	MOV	DL,BL
3985:0104 80C230	ADD	DL,30
3985:0107 80FA3A	CMP	DL,3A
3985:010A 7C03	JL	010F
3985:010C 80C207	ADD	DL,07
3985:010F CD21	INT	21
3985:0111 CD20	INT	20

0107 CMP DL,3A

010A JL 010F

010C ADD DL,07

010F INT 21



Afb. 5-2. De JL-instructie.

De CMP-instructie trekt, zoals we zagen, twee getallen van elkaar af (DL – 3Ah) om de vlaggen te zetten, maar verandert DL niet. Dus als DL kleiner is dan 3Ah, springt de instructie JL 010F naar de INT 21h op 010Fh. Zet nu een hex-getal van één cijfer in BL en traceer dit voorbeeld om vertrouwd te raken met de CMP-instructie en een beter idee te krijgen van ons algoritme voor het omzetten van hex in ASCII. Vergeet niet de G-opdracht met een breakpoint of de P-opdracht te gebruiken wanneer u de INT-instructie tegenkomt.

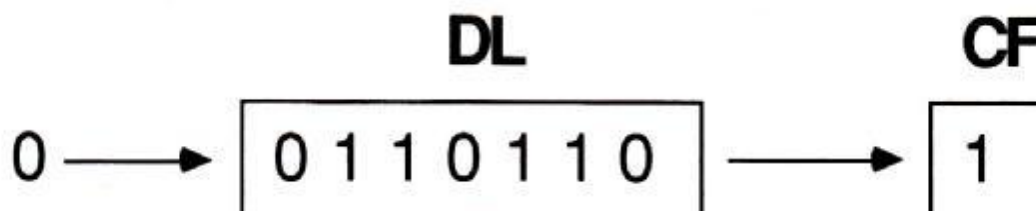
5.3 Nog een roteer-instructie

Ons programma lukt bij elk hex-getal van één cijfer, maar als we een hex-getal van twee cijfers willen afdrukken, hebben we nog een paar stappen nodig. We moeten elk cijfer (vier bits, vaak *nibble* genoemd) van dit tweecijferige getal isoleren. In deze paragraaf zullen we zien dat we gemakkelijk de vier eerste, of hoogste, bits kunnen isoleren, en in de volgende paragraaf komen we een begrip tegen, aangeduid met de naam *logische bewerking*, dat we zullen gebruiken voor het isoleren van de vier laagste bits — het tweede van onze twee hex-cijfers.

Om te beginnen herinneren we eraan dat de RCL-instructie een byte of een woord naar links roteert, via de overdrachtvlag. In het vorige hoofdstuk hebben we de instructie RCL BL,1 gebruikt, waarin de 1 betekende dat de 8088 BL één bit moest roteren. We kunnen desgewenst met meer dan één bit roteren, maar we kunnen dat niet gewoon doen met de instructie RCL BL,2. (N.B. Hoewel RCL BL,2 geen toegestane 8088-instructie is, werkt hij wel bij de 80286-processor die je in IBM-AT's aantreft. Maar omdat de oudere IBM-PC's en XT's meer voorkomen dan de AT's, kunt u uw programma's het beste voor de grootste gemene deler — de oudere 8088 — schrijven.) Voor rotaties met meer dan één bit moeten we een roteerteller in het CL-register zetten.

Het CL-register wordt hier op vrijwel dezelfde manier gebruikt als het CX-register door de LOOP-instructie wordt gebruikt om het aantal keren te bepalen dat een lus moet worden herhaald. Gebruik liever CL dan CX voor het aantal keren dat een byte of woord moet worden geroteerd, omdat het geen zin heeft om meer dan zestien keer te roteren; het CL-register is daarom groot genoeg om het getal voor het maximum aantal verschuivingen te bevatten.

Wat heeft dit nu allemaal te maken met het afdrukken van een hex-getal van twee cijfers? Wat we willen, is de byte in DL vier bits naar rechts roteren. Daartoe zullen we een iets andere roteer-instructie gebruiken, genaamd SHR (*Shift Right*, verschuif naar rechts). Met SHR kunnen we de vier hoogste bits van ons getal naar de meest rechtse nibble (vier bits) verplaatsen.



Afb. 5-3. De instructie SHR DL,1.

We willen ook de vier hoogste bits van DL nul maken zodat het hele register gelijk wordt aan de byte die we in de rechter nibble zetten. Als we SHR DL,1 zouden gebruiken, zou onze instructie de byte in DL één bit naar rechts verplaatsen, en *tegelijk* bit 0 in de overdrachtvlag zetten, terwijl hij een nul in bit 7 (het hoogste, of meest linkse, bit in DL) zou schuiven. Als we dat dan nog drie keer doen, hebben we precies wat we willen. De bovenste vier bits zijn dan ten slotte naar de vier laagste bits geschoven, terwijl in de bovenste vier bits allemaal nullen staan. We kunnen al die verschuivingen tot stand brengen met één instructie, en het CL-register daarbij gebruiken als *schuif*-teller. Door CL voor de instructie SHR DL,CL op vier te zetten, zorgen we ervoor dat DL gelijk wordt aan het bovenste hex-cijfer.

Laten we eens kijken hoe dat gaat. Zet 4 in CL en 5Dh in DL, voer dan de volgende SHR-instructie in en traceer die:

```
3985:0100 D2EA          SHR    DL,CL
```

DL moet nu 05h zijn, wat het eerste cijfer van het getal 5Dh is, en we kunnen dit cijfer nu afdrukken met een programma zoals we al eerder hebben gebruikt. Door de stukjes die we tot nu toe hebben, bijeen te leggen, kunnen we het volgende programma maken, dat een getal in het BL-register neemt en daar het eerste hex-cijfer van afdrukt:

```
3985:0100 B402          MOV    AH,02
3985:0102 88DA          MOV    DL,BL
3985:0104 B104          MOV    CL,04
3985:0106 D2EA          SHR    DL,CL
3985:0108 80C230        ADD    DL,30
3985:010B 80FA3A        CMP    DL,3A
3985:010E 7C03          JL     0113
3985:0110 80C207        ADD    DL,07
3985:0113 CD21          INT    21
3985:0115 CD20          INT    20
```

5.4 Logica en AND

Nu we het eerste van de twee cijfers van een hex-getal kunnen afdrukken, gaan we eens zien hoe we het tweede cijfer kunnen isoleren en afdrukken. We leren in deze paragraaf hoe we de vier hoogste bits van onze oorspronkelijk (niet verschoven) getal op nul kunnen zetten, en DL daarbij gelijkhouden aan de vier laagste bits. Dat is eenvoudig: zet de vier hoogste bits op nul met een instructie genaamd AND. De AND-instructie is één van de logische instructies, die hun oorsprong vinden in de formele, mathematische logica. Laten we eens kijken hoe AND werkt.

In de formele logica kunnen we zeggen: 'A is waar als B en C beide waar zijn'. Maar als óf B óf C onwaar is, dan moet A ook onwaar zijn. Als we dit nu oppakken en *waar* door 1 en *onwaar* door 0 vervangen en vervolgens kijken naar de verschillende combinaties van A, B en C, kunnen we een waarheidstabel opstellen, zoals dat heet. Dit is de waarheidstabel voor het ANDen van twee bits bij elkaar:

AND	F	T		AND	0	1
F	F	F	=	0	0	0
T	F	T		1	0	1

T staat hierbij voor *True* (waar) en *F* voor *False* (onwaar). Aan de linkerkant en bovenlangs staan de waarden van de twee bits. De uitkomsten van de AND staan in de tabel, dus u ziet dat bijvoorbeeld 0 AND 1 een resultaat 0 oplevert.

De AND-instructie werkt ten aanzien van bytes en woorden door de bits van elke byte of elk woord die op dezelfde positie staan te ANDen. Zo ANDt de opdracht AND BL,CL achtereenvolgens de bits 0 van BL en CL, de bits 1, de bits 2, enzovoort, en zet het resultaat in BL. We zullen dit verduidelijken met een voorbeeld in binair:

	1	0	1	1	0	1	0	1
AND	0	1	1	1	0	1	1	0
	0	0	1	1	0	1	0	0

Verder kunnen we door 0Fh met elk ander getal te ANDen de vier hoogste bits op nul zetten:

	0	1	1	1	1	0	1	1
AND	0	0	0	0	1	1	1	1
	0	0	0	0	1	0	1	1

Laten we deze logica nu eens toepassen in een programmaatje dat het getal in BL pakt, het laagste hex-cijfer isoleert door 0Fh te ANDen met de vier hoogste bits, en het resultaat dan als een teken af te drukken. De meeste onderdelen van dit programma hebben we gezien toen we het hoogste hex-cijfer afdrukten; het enige nieuwe is de AND-instructie.

3985:0100 B402	MOV	AH,02
3985:0102 88DA	MOV	DL,BL
3985:0104 80E20F	AND	DL,0F
3985:0107 80C230	ADD	DL,30
3985:010A 80FA3A	CMP	DL,3A
3985:010D 7C03	JL	0112
3985:010F 80C207	ADD	DL,07
3985:0112 CD21	INT	21
3985:0114 CD20	INT	20

Probeer dit eens met hex-getallen van twee cijfers in BL voor we verder gaan om de stukken bij elkaar te leggen en beide cijfers af te drukken. U moet het meest rechtse hex-cijfer van uw oorspronkelijke getal in BL dan op het scherm zien.

5.5 Alles bij elkaar leggen

Er hoeft eigenlijk niet veel te worden veranderd wanneer we alle stukjes bij elkaar leggen. We moeten alleen het adres veranderen van de tweede JL-instructie die we gebruikten om het tweede hex-cijfer af te drukken. Dit is het volledige programma:

3985:0100	B402	MOV	AH,02
3985:0102	88DA	MOV	DL,BL
3985:0104	B104	MOV	CL,04
3985:0106	D2EA	SHR	DL,CL
3985:0108	80C230	ADD	DL,30
3985:010B	80FA3A	CMP	DL,3A
3985:010E	7C03	JL	0113
3985:0110	80C207	ADD	DL,07
3985:0113	CD21	INT	21
3985:0115	88DA	MOV	DL,BL
3985:0117	80E20F	AND	DL,0F
3985:011A	80C230	ADD	DL,30
3985:011D	80FA3A	CMP	DL,3A
3985:0120	7C03	JL	0125
3985:0122	80C207	ADD	DL,07
3985:0125	CD21	INT	21
3985:0127	CD20	INT	20

Hebt u dit programma ingevoerd, tik dan *U 100*, gevolgd door *U*, om de hele gedisassembleerde listing te zien. Merk op dat we een groep van vijf instructies hebben herhaald: die van 108h tot en met 113h, en van 11Ah tot en met 125h. In hoofdstuk 7 zullen we zien hoe we een dergelijke reeks instructies maar één keer hoeven te schrijven met behulp van eenzelfde soort instructie als de GOSUB-opdracht in BASIC.

5.6 Samenvatting

In dit hoofdstuk hebben we meer geleerd over de manier waarop Debug getallen van de binaire 8088-vorm vertaalt in een hex-vorm die we kunnen lezen. Wat hebben we allemaal aan onze groeiende kennis toegevoegd?

Eerst hebben we enkele tweelettervlaggen behandeld, die je ziet aan de rechterkant van de register (R)-afbeelding. Deze statusbits geven ons veel informatie over onze laatste rekenkundige bewerking. Door bijvoorbeeld naar de nulvlag te kijken, konden we zien of het resultaat van de laatste bewerking nul was. Ook leerden we dat we twee getallen konden vergelijken met een CMP-instructie.

Daarna leerden we hoe we een hex-getal van een cijfer konden afdrukken. En gewapend met die kennis leerden we vervolgens de SHR-instructie gebruiken om het hoogste cijfer van een tweecijferig hex-getal in de vier laagste bits van BL te zetten. Daarna konden we het cijfer afdrukken, net als we eerder hadden gedaan.

Ten slotte zagen we dat met de AND-instructie het laagste hex-cijfer van het hoogste kan worden gescheiden. En door al die stukjes bij elkaar te leggen, konden we een programma schrijven dat een hex-getal van twee cijfers afdrukt.

We hadden verder kunnen gaan met het afdrukken van een hex-getal van vier cijfers, maar in dit stadium zouden we dan alleen maar instructies herhalen. Voor we een viercijferig hex-getal proberen af te drukken, leren we in hoofdstuk 7 over procedures. Daarna weten we voldoende om een procedure te schrijven die het werk voor ons doet. We zijn dan ook klaar om iets over de *Assembler* te leren — een programma dat ons veel werk uit handen zal nemen. Maar nu gaan we eerst hex-getallen lezen.

6 Tekens lezen

- 6.1 Een teken lezen 76**
- 6.2 Een hex-getal van een cijfer lezen 76**
- 6.3 Een hex-getal van twee cijfers lezen 77**
- 6.4 Samenvatting 78**

Nu we weten hoe een byte in de hex-notatie kan worden afgedrukt, gaan we het omgekeerde doen en twee tekens — hex-cijfers — van het toetsenbord lezen en in één byte omzetten.

6.1 Een teken lezen

De DOS-INT 21h-interrupt die we hebben gebruikt, heeft een invoerfunctie, nummer 1, waarmee een teken van het toetsenbord wordt gelezen. Toen we in hoofdstuk 4 leerden over functie-aanroepen, zagen we dat het functienummer voor een INT 21h-aanroep in het AH-register moet worden gezet. Laten we functie 1 van INT 21h eens proberen. Zet INT 21h op geheugenplaats 0100h:

396F:0100 CD21

INT 21

Zet dan 01h in AH en tik óf *G 012* of *P* om deze ene instructie uit te voeren. Gebeurt er niets? Dat lijkt zo — het enige wat u ziet is de knipperende cursor. Maar in feite is DOS gestopt en wacht het tot u een toets indrukt (nog niet doen). Als u eenmaal een toets hebt ingedrukt, zet DOS de ASCII-code van dat teken in het AL-register. We zullen deze instructie verderop gebruiken om de tekens van een hex-getal te lezen, maar laten we eerst even kijken wat er gebeurt als we op een toets als F1 drukken. Druk de toets F1 nu eens in. DOS zet dan een 0 in AL, en u zult vlak achter het prompt-streepje van Debug ook een puntkomma op het scherm zien verschijnen. Wat er gebeurt, is het volgende. F1 is een van de speciale toetsen met codes van de *uitgebreide tekenset*, die DOS anders behandelt dan de toetsen die de gewone ASCII-tekens voorstellen. (In appendix E, en ook achterin uw BASIC-handleiding, staan deze uitgebreide codes samengevat in een tabel.) Voor elk van deze toetsen verstuurt DOS *twee* tekens, het ene vlak na het andere. Het eerste teken dat wordt doorbestuurd is altijd een nul, om aan te geven dat het volgende teken de *scan-code* voor een speciale toets is.

Om beide tekens te lezen, moeten we de INT 21h twee keer uitvoeren. Maar in ons voorbeeld hebben we alleen het eerste teken, de nul, gelezen en de scan-code laten zitten. Toen Debug klaar was met de opdracht *G 102* (of *P*), begon het tekens te lezen, en het eerste teken dat het las, was de scan-code die nog van de F1-toets was overgebleven: 59, de ASCII-code voor een puntkomma.

Verderop, bij het ontwikkelen van ons Dskpatch-programma, zullen we deze uitgebreide codes gebruiken om de cursor- en functietoetsen tot leven te brengen. Voorlopig doen we het met de gewone ASCII-tekens.

6.2 Een hex-getal van een cijfer lezen

We gaan de omzetting die we hebben gebruikt in hoofdstuk 5, toen we een hex-getal van één cijfer omzetten in de ASCII-code voor een van de tekens in 0 t/m 9 of A t/m F, nu omdraaien. Om een teken, als C of D, om te zetten van een hex-teken in een byte, moeten we er óf 30h (voor 0 t/m 9) óf 37h (voor A t/m F) van aftrekken. Hier is een eenvoudig programma dat een ASCII-teken leest en omzet in een byte:

3985:0100	B401	MOV	AH,01
3985:0102	CD21	INT	21
3985:0104	2C30	SUB	AL,30
3985:0106	3C09	CMP	AL,09
3985:0108	7E02	JLE	010C
3985:010A	2C07	SUB	AL,07
3985:010C	CD20	INT	20

De meeste instructies hierin zullen u nu wel bekend voorkomen, maar er is een nieuwe bij: JLE (*Jump if Less than or Equal to*, spring indien kleiner dan of gelijk aan). In ons programma springt deze instructie indien AL kleiner dan of gelijk aan 9h is. Om de omzetting van hex-teken naar ASCII te kunnen zien, moet u vlak voor uitvoering van de INT 20h het AL-register bekijken. Omdat Debug de registers herstelt wanneer hij de INT 20h uitvoert, moet u een breakpoint instellen, net als in hoofdstuk 4. Hier tikt u *G 10C*, en u zult zien dat AL het hex-getal bevat dat vanuit een teken is omgezet.

Probeer eens wat tekens te tikken, die geen hex-cijfers zijn, zoals *k* of een kleine letter *d*, en kijk wat er gebeurt. Het zal u opvallen dat dit programma alleen goed werkt wanneer het ingetikte teken een van de cijfers 0 t/m 9 of de hoofdletters A t/m F is. We zullen dit kleine euvel in het volgende hoofdstuk verhelpen, wanneer we subroutines, of procedures, bespreken. Voorlopig zijn we even wat slordig en slaan geen acht op foutcondities: we zullen de juiste tekens moeten tikken om ons programma goed te laten werken.

6.3 Een hex-getal van twee cijfers lezen

Twee hex-cijfers lezen is niet veel ingewikkelder dan één, maar er zijn wel veel meer instructies voor nodig. We beginnen met het eerste cijfer te lezen, vervolgens zetten we de hex-waarde daarvan in het DL-register en vermenigvuldigen dat met 16. Voor deze vermenigvuldiging schuiven we het DL-register vier bits naar links en zetten een hex-nul (vier nul-bits) rechts van het cijfer dat we net hebben gelezen. De instructie SHL DL,CL, waarbij CL vier is, doet dit door nullen aan de rechterkant in te voegen. De SHL (*Shift Left*, verschuif naar links)-instructie staat bekend als een *rekenkundige verschuiving* omdat ze hetzelfde effect heeft als een rekenkundige vermenigvuldiging met twee, vier, acht, enz., afhankelijk van het getal (een, twee, drie enz.) in CL.



Afb. 6-1. De instructie SHL DL,1.

Nu het eerste cijfer verplaatst is, zullen we ten slotte nog het tweede hex-cijfer optellen bij het getal in DL (het eerste cijfer * 16). U kunt al deze details in het volgende programma bekijken en zien hoe het werkt:

3985:0100	B401	MOV	AH,01
3985:0102	CD21	INT	21
3985:0104	88C2	MOV	DL,AL
3985:0106	80EA30	SUB	DL,30
3985:0109	80FA09	CMP	DL,09
3985:010C	7E03	JLE	0111
3985:010E	80EA07	SUB	DL,07
3985:0111	B104	MOV	CL,04
3985:0113	D2E2	SHL	DL,CL
3985:0115	CD21	INT	21
3985:0117	2C30	SUB	AL,30
3985:0119	3C09	CMP	AL,09
3985:011B	7E02	JLE	011F
3985:011D	2C07	SUB	AL,07
3985:011F	00C2	ADD	DL,AL
3985:0121	CD20	INT	20

Nu we een werkend programma hebben, is het verstandig om de grenscondities te controleren om te zien of het ook goed werkt. Neem voor deze condities de getallen 00, 09, 0A, 90, A0, F0 en nog een ander getal, zoals 3C. Gebruik een breakpoint om het programma te draaien zonder de INT 20h-instructie uit te voeren. (Zorg dat u hoofdletters gebruikt voor de hex-invoer.)

6.4 Samenvatting

We hebben eindelijk de kans gehad om het geleerde van de vorige hoofdstukken in praktijk te brengen zonder met nieuwe informatie te worden overstelpt. Door een nieuwe INT 21h-functie (nummer 1) te gebruiken, hebben we een programma ontwikkeld dat een hex-getal van twee cijfers leest. In het voorbijgaan hebben we er met nadruk op gewezen dat programma's met alle grenscondities moeten worden getest. We kunnen deel 1 nu afsluiten met een bespreking van procedures in de 8088.

7 Procedures — verwant met subroutines

- 7.1 Procedures 80**
- 7.2 De stapel en terugkeeradressen 81**
- 7.3 PUSH en POP 83**
- 7.4 Gemakkelijker hex-getallen lezen 85**
- 7.5 Samenvatting 87**

In het volgende hoofdstuk maken we kennis met MASM, de macro-assembler, en gaan we aan het werk met de *assembleertaal*. Maar vóór we Debug verlaten, bekijken we nog wat laatste voorbeelden, en leren we over subroutines en een speciale plaats voor het opslaan van gegevens, de *stapel*.

7.1 Procedures

Een procedure is een reeks instructies die we vanuit allerlei verschillende plaatsen in een programma kunnen uitvoeren in plaats van dezelfde instructies te moeten herhalen op elke plaats waar ze nodig zijn. In BASIC worden zulke reeksen subroutines genoemd, maar wij noemen ze *procedures* om redenen die later duidelijk zullen worden.

We gaan van en naar procedures net zoals we dat in BASIC doen. We roepen een procedure aan met een instructie, *CALL*, die precies zo werkt als de BASIC-opdracht GOSUB. En we keren uit de procedure terug met de instructie *RET*, die precies zo werkt als de BASIC-opdracht RETURN.

Hieronder staat een BASIC-programma dat we straks in machinetaal zullen herschrijven. Dit programma roept tien keer een subroutine aan, en drukt elke keer een teken af, te beginnen met A en dan verder tot aan J:

```
10 A = &H41          'ASCII voor 'A'
20 FOR I = 1 TO 10
30 GOSUB 1000
40 NEXT I
50 END
1000 PRINT CHR$(A);
1100 A = A + 1
1200 RETURN
```

De subroutine begint, zoals wel gebruikelijk is in BASIC-programma's, op regel 1000 om ruimte open te laten zodat we instructies aan het hoofdprogramma kunnen toevoegen zonder dat onze subroutine in de weg staat. We zullen hetzelfde doen bij onze machinetaal-procedure door die op 200h te zetten, ver weg van ons hoofdprogramma op 100h. Ook zullen we GOSUB 1000 vervangen door de instructie CALL 200h, die de procedure op de geheugenplaats 200h aanroept. De CALL zet IP op 200h, en de 8088 begint de instructies op 200h uit te voeren.

De FOR-NEXT-lus van het BASIC-programma kan, zoals we in hoofdstuk 4 zagen, als een LOOP-instructie worden geschreven. De andere delen van het programma moeten u bekend zijn.

3985:0100 B241	MOV	DL,41
3985:0102 B90A00	MOV	CX,000A
3985:0105 E8F800	CALL	0200
3985:0108 E2FB	LOOP	0105
3985:010A CD20	INT	20

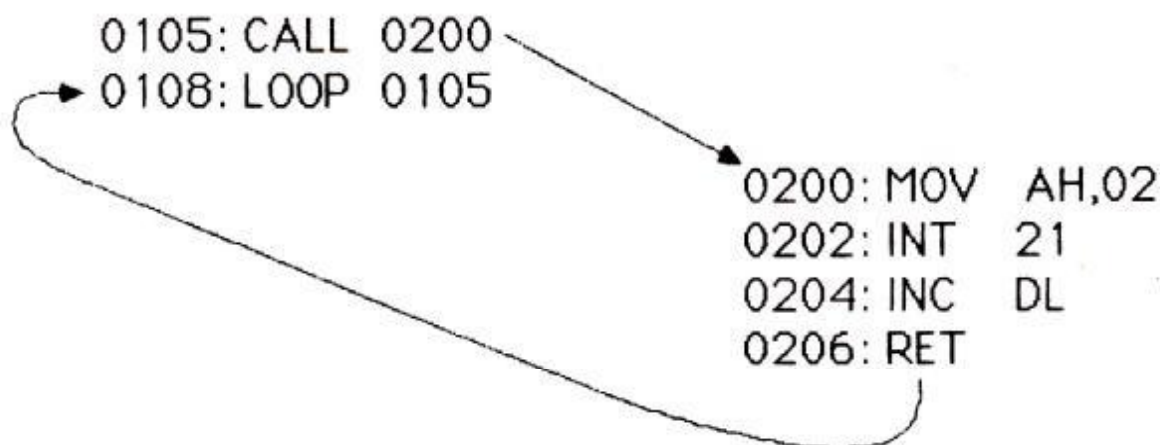
De eerste instructie zet 41h (ASCII voor A) in het DL-register omdat de instructie INT 21h-instructie het teken afdrukt dat met de ASCII-code is aangeduid in DL.

De INT 21h-instructie zelf staat een eind verderop, in de procedure op geheugenplaats 200h. Dit is de procedure die u op 200h moet invoeren:

```
3985:0200 B402      MOV     AH,02
3985:0202 CD21      INT      21
3985:0204 FEC2      INC      DL
3985:0206 C3       RET
```

Hier staan twee nieuwe en twee oude instructies in. U zult nog weten dat de 02h in AH DOS vertelt dat het teken in DL moet worden afgedrukt wanneer we de INT 21h-instructie uitvoeren. INC DL (van *increment*, verhogen), de eerste van de twee nieuwe instructies, hoogt het DL-register op. Dat wil zeggen telt één bij DL op. De andere nieuwe instructie, RET (van *return*, terugkeren), keert terug naar de eerste (LOOP)-instructie na de CALL in ons hoofdprogramma.

Tik G om de uitvoer van dit programma te zien, neem het dan met één stap tegelijk door om te zien hoe het werkt (vergeet niet een breakpoint te zetten of de P-opdracht te gebruiken voor de INT 21h-instructie).

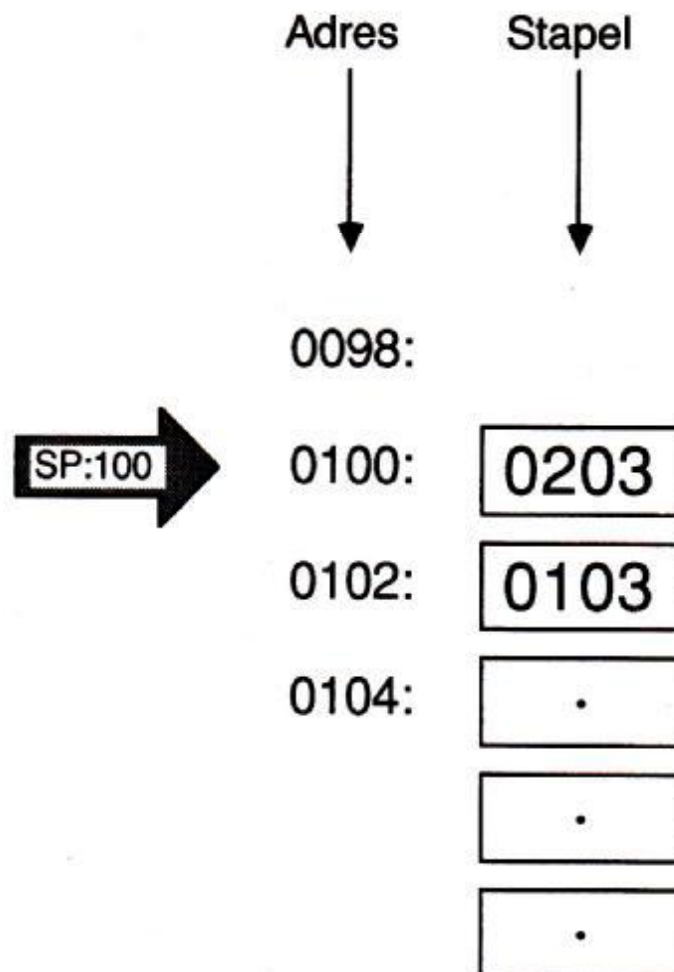


Afb. 7-1. De instructies CALL en RET.

7.2 De stapel en terugkeeradressen

De CALL-instructie in ons programma moet ergens het terugkeeradres opslaan zodat de 8088 weet waar hij verder moet gaan met het uitvoeren van de instructies wanneer hij de RET-instructie ziet. Voor de opslagruimte zelf beschikken we over een deel van het geheugen dat bekend staat als *stack* of stapel. En om te kunnen nagaan wat er op de stapel staat, zijn er twee registers die we in de R-afbeelding van Debug kunnen bekijken: het SP (*Stack Pointer*, stapelwijzer)-register, dat naar de top van de stapel wijst, en het SS (*Stack Segment*, stapelsegment)-register, dat het segmentnummer bevat.

Een stapel werkt voor de 8088 ongeveer als een bordenautomaat in een restaurant, waar je borden bovenop een stapel andere borden zet. Het laatste bord op de stapel wordt er het eerst weer afgehaald, daarom wordt een stapel ook wel aangeduid met



Afb. 7-2. De stapel vlak voor uitvoering van de instructie CALL 400.

het woord LIFO (*Last In First Out*, laatste erin, eerste eruit). Deze LIFO-volgorde is precies wat we nodig hebben voor het opvragen van terugkeeradressen bij *geneste* CALLs zoals deze:

396F:0100 E8FD00	CALL	0200
		.
		.
396F:0200 E8FD00	CALL	0300
396F:0203 C3	RET	
		.
		.
396F:0300 E8FD00	CALL	0400
396F:0303 C3	RET	
		.
		.
396F:0400 C3	RET	

De instructie op 100h roept hier een procedure op 200h aan, die er een op 300h aanroept, die er een op 400h aanroept, waar we ten slotte een terugkeer (RET)-instructie zien. Deze RET zorgt voor terugkeer naar de instructie volgend op de *vorige* CALL-instructie, op 300h, zodat de 8088 het uitvoeren van instructies hervat op 303h. Maar daar komt hij op 303h een RET-instructie tegen, die het op één na oudste adres (203h) van de stapel haalt. Dus hervat de 8088 het uitvoeren van de instructies op 203h, enz. Elke RET haalt het hoogste terugkeeradres van de stapel, zodat elke RET dezelfde weg terug volgt als de CALLs naar voren volgden.

Tik nu eens een programma als hierboven in. Gebruik meerdere aanroepen en traceer het programma om te zien hoe het aanroepen en terugkeren werkt. Hoewel het u misschien op dit moment allemaal niet zo interessant lijkt, wordt de stapel nog op andere manieren gebruikt, en een goed idee van hoe hij werkt kan van pas komen. (In een volgend hoofdstuk zoeken we de stapel in het geheugen op.)

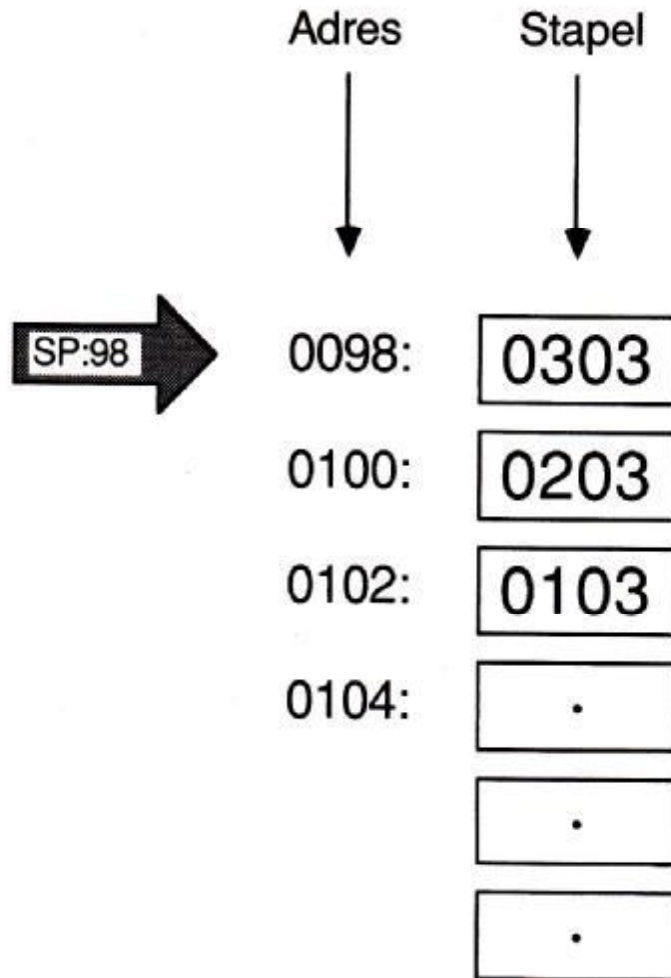
7.3 PUSH en POP

De stapel is een nuttige plaats om enige tijd woorden en gegevens op te slaan, mits we ervoor zorgen dat de stapel wordt hersteld vóór een RET-instructie. We hebben gezien dat een CALL-instructie het terugkeeradres (een woord) bovenop de stapel duwt, terwijl een RET-instructie dat woord van de top van de stapel naar het IP-register brengt, zodat het woord eronder bovenaan komt. We kunnen vrijwel hetzelfde doen met de instructie PUSH en POP, waarmee we woorden op de stapel kunnen duwen (*push*) en eraf halen (*pop*). Wanneer zouden we dat willen?

Het is vaak gemakkelijk om aan het begin van een procedure de waarden van registers te bewaren en ze aan het eind, vlak voor de RET-instructie, terug te zetten. Op die manier kunnen we deze registers vrijelijk op elke gewenste manier binnen de procedure gebruiken, zo lang we hun waarden aan het eind maar weer terugzetten. Programma's bestaan uit procedures op tal van niveaus, waarbij op elk niveau de procedures van het niveau eronder worden aangeroepen. Door registers aan het begin van een procedure te bewaren en ze aan het eind te herstellen, voorkomen we ongewenste interacties tussen procedures op verschillende niveaus, wat ons het programmeren een stuk gemakkelijker maakt. In hoofdstuk 13 zult u meer over het bewaren en herstellen van registers zien, wanneer we het over modulair ontwerpen hebben. Voorlopig geven we alleen een voorbeeld (niet invoeren) waarin CX en DX worden bewaard en hersteld:

396F:0200 51	PUSH	CX
396F:0201 52	PUSH	DX
396F:0202 B90800	MOV	CX,0008
396F:0205 E8F800	CALL	0300
396F:0208 FEC2	INC	DL
396F:020A E2F9	LOOP	0205
396F:020C 5A	POP	DX
396F:020D 59	POP	CX
396F:020E C3	RET	

Merk op dat de POPs in de omgekeerde volgorde van de PUSHes staat, omdat een POP het woord dat het laatst op de stapel is gezet eraf haalt, en de oude waarde van DX op de oude van CX staat.



Afb. 7-3. De stapel vlak na uitvoering van de instructie CALL 400.

Door CX en DX te bewaren en terug te zetten, kunnen we deze registers veranderen in de procedure die op 200h begint, maar zonder de waarden te veranderen die door een procedure worden gebruikt die deze aanroept. En als we eenmaal CX en DX hebben bewaard, kunnen we deze registers gebruiken voor het vasthouden van *lokale* variabelen — variabelen die we kunnen gebruiken binnen deze procedure zonder dat dat van invloed is op de waarden die door het aanroepende programma worden gebruikt.

We zullen dergelijke lokale variabelen gebruiken om het ons bij het programmeren gemakkelijker te maken. Zo lang we zorgen dat de oorspronkelijke waarden worden teruggezet, hoeven we niet bang te zijn dat onze procedures registers veranderen die door het aanroepende programma worden gebruikt. Dit wordt duidelijker in het volgende voorbeeld, dat bestaat uit een procedure waarmee een hex-getal wordt gelezen. Anders dan met het programma in hoofdstuk 6, kunnen met dit programma alleen geldige tekens zoals A, en niet K, worden gelezen.

7.4 Gemakkelijker hex-getallen lezen

We willen een procedure maken die tekens blijft lezen tot hij er een krijgt die hij in een hex-getal tussen 0 en Fh kan omzetten. We willen geen ongeldige tekens afbeelden, dus we zullen onze invoer selecteren door gebruik te maken van een nieuwe INT 21h-functie, nummer 8, die een teken leest maar niet doorstuurt naar het scherm. Op die manier kunnen we tekens alleen naar het scherm *echoën* (erop afbeelden) wanneer ze geldig zijn.

Zet 8h in het AH-register en neem deze instructie door, door na het tikken van G 102 een A te tikken.

```
3985:0100 CD21          INT    21
```

De ASCII-code voor A (41h) staat nu in het AL-register, maar de A is niet op het scherm verschenen.

Met deze functie kan ons programma tekens lezen zonder ze te echoën tot het een geldig hex-cijfer (0 t/m 9 of A t/m F) leest dat het dan op het scherm afbeeldt. De procedure daarvoor en voor het omzetten van het hex-teken in een hex-getal ziet er als volgt uit:

```
3985:0200 52          PUSH    DX
3985:0201 B408        MOV     AH,08
3985:0203 CD21        INT     21
3985:0205 3C30        CMP     AL,30
3985:0207 72FA        JB      0203
3985:0209 3C46        CMP     AL,46
3985:020B 77F6        JA      0203
3985:020D 3C39        CMP     AL,39
3985:020F 770A        JA      021B
3985:0211 B402        MOV     AH,02
3985:0213 88C2        MOV     DL,AL
3985:0215 CD21        INT     21
3985:0217 2C30        SUB     AL,30
3985:0219 5A         POP      DX
3985:021A C3         RET
3985:021B 3C41        CMP     AL,41
3985:021D 72E4        JB      0203
3985:021F B402        MOV     AH,02
3985:0221 88C2        MOV     DL,AL
3985:0223 CD21        INT     21
3985:0225 2C37        SUB     AL,37
3985:0227 5A         POP      DX
3985:0228 C3         RET
```

De procedure leest een teken in AL (met de INT 21h op 203h) en controleert met de CMP-instructies en voorwaardelijke sprongen of het geldig is. Als het net gelezen teken geen geldig teken is, sturen de voorwaardelijke spronginstructies de 8088 terug naar geheugenplaats 203, waar de INT 21h een nieuw teken leest. (JA betekent *Jump if Above* (spring indien hoger) en JB *Jump if Below* (spring indien lager); beide behandelen de twee getallen als getallen zonder teken, terwijl de JL-instructie die we eerder gebruikten beide als getal met teken behandelde.)

Op regel 211h weten we dat we een geldig cijfer tussen 0 en 9 hebben, dus we trekken

de code voor het teken nul eraf en zetten het resultaat in het AL-register, en vergeten niet het DX-register, dat we aan het begin van de procedure hadden bewaard, van de stapel te halen. Bij de hex-cijfers A t/m F gaat het vrijwel net zo. Merk op dat we twee RET-instructies in deze procedure hebben; we hadden er meer kunnen hebben, of maar één.

Dit is een eenvoudig programmaatje om de procedure te testen:

```
3985:0100 E8FD00      CALL    0200
3985:0103 CD20        INT     20
```

Gebruik, zoals u al eerder hebt gedaan, de G-opdracht met een breakpoint, of de P-opdracht. U moet wel de CALL 200h-instructie zónder de instructie INT 20h uitvoeren, om de registers te kunnen zien vlak voor het programma stopt en de registers hersteld worden.

U zult de cursor aan de linkerkant van het scherm zien staan, geduldig wachtend tot er een teken wordt getikt. Tik eens *k*, dat geen geldig teken is. Er mag dan niets gebeuren. Tik nu een van de hex-hoofdletters. U moet dan de hexwaarde van het teken in AL zien en het teken zelf op het scherm. Test deze procedure met de grenscondities: '/' (het teken vóór de nul), 0, 9, ':' (het teken vlak na de 9) enz.

Nu we deze procedure hebben, is het programma voor het lezen van hex-getal van twee cijfers, met foutafhandeling, vrij simpel:

```
3985:0100 E8FD00      CALL    0200
3985:0103 88C2        MOV     DL,AL
3985:0105 B104        MOV     CL,04
3985:0107 D2E2        SHL     DL,CL
3985:0109 E8F400      CALL    0200
3985:010C 00C2        ADD     DL,AL
3985:010E B402        MOV     AH,02
3985:0110 CD21        INT     21
3985:0112 CD20        INT     20
```

U kunt dit programma vanuit DOS draaien, omdat het een tweecijferig hex-getal leest en dan het ASCII-teken afdrukt dat overeenkomt met het getal dat u net hebt ingetikt.

Afgezien van deze procedure, is ons hoofdprogramma veel eenvoudiger dan de versie die we in het vorige hoofdstuk hebben geschreven, en we hebben de instructies voor het lezen van tekens niet herhaald. We hebben er echter wel een foutafhandeling aan toegevoegd, en dat mag dan onze procedure ingewikkeld hebben gemaakt, maar het zorgt er ook voor dat het programma alleen geldige invoer accepteert.

We kunnen hier ook zien waarom het DX-register wordt bewaard. Het hoofdprogramma slaat het hex-getal op in DL, dus we willen niet dat onze procedure op 200h DL verandert. Aan de andere kant moet de procedure op 200h DL zelf gebruiken om tekens naar het scherm te sturen. Door nu de instructie PUSH DX aan het begin van de procedure te zetten, en POP DX aan het eind, besparen we ons problemen. Om ingewikkelde interacties tussen procedures te voorkomen, zullen we van nu af heel nauwgezet registers bewaren die door een procedure worden gebruikt.

7.5 Samenvatting

Onze procedures worden al verfijnder. We hebben geleerd over procedures waarmee we dezelfde reeks instructies opnieuw kunnen gebruiken zonder ze steeds weer te moeten schrijven. We hebben ook de stapel ontdekt en gezien dat een CALL een terugkeeradres bovenop de stapel zet, terwijl een RET-instructie zorgt voor terugkeer naar het adres op de top van de stapel.

We hebben gezien hoe de stapel wordt gebruikt voor meer dan alleen het bewaren van terugkeeradressen. We hebben de stapel gebruikt om de waarden van registers te bewaren (met een PUSH-instructie) om die in procedure te kunnen gebruiken. Door de registers aan het eind van elke procedure weer te herstellen (met een POP-instructie), hebben we ongewenste interacties tussen procedure voorkomen. Door altijd registers in procedures die we schrijven te bewaren en terug te zetten, kunnen we andere procedures met CALL aanroepen zonder ons te hoeven bekommeren over welke registers binnen de andere procedure worden gebruikt.

En ten slotte vervolgden we, gewapend met deze kennis, onze weg en schreven een beter programma om hex-getallen in te lezen — deze keer met foutcontrole. Het programma dat we hier hebben opgezet lijkt op een dat we in latere hoofdstukken zullen gebruiken wanneer we het Dskpatch-programma ontwikkelen.

Nu zijn we klaar om door te gaan naar deel 2, waarin we de assembler zullen leren gebruiken. In het volgende hoofdstuk zullen we zien hoe de assembler kan worden gebruikt om een programma om te zetten in machinetaal. We zullen ook zien dat er geen enkele reden bestaat om ruimte tussen procedures te zetten, zoals we in dit hoofdstuk deden, toen we onze procedure ver weg op de geheugenplaats 200h lieten beginnen.

DEEL 2

Assembleertaal

8 Welkom bij de assembler

8.1	Een programma zonder Debug	92
8.2	Aanmaken van bronbestanden	95
8.3	Linken	95
8.4	Terug naar Debug	97
8.5	Commentaar	97
8.6	Labels	98
8.7	Samenvatting	100

Eindelijk is dan het moment gekomen om kennis te maken met de assembler, een programma dat ons het programmeren veel gemakkelijker gaat maken. Van nu af zullen we rechtstreeks mnemonische, door mensen leesbare, instructies schrijven en de assembler gebruiken om onze programma's om te zetten in machinetaal.

In dit en het volgende hoofdstuk zal er nogal in details over de assembler worden gesproken, maar het zal de moeite waard blijken om die onder de knie te krijgen. Als we de assembler eenmaal weten te gebruiken, gaan we terug naar waar we mee bezig waren: programma's in assembleertaal schrijven. We beginnen meteen maar.

8.1 Een programma zonder Debug

Tot dusver hebben we alleen *DEBUG* getikt, en daarna onze programma-instructies ingevoerd. Nu gaan we Debug achter ons laten en programma's zonder DEBUG schrijven, en we zullen een editor of een tekstverwerker moeten gebruiken om tekst-, of door mensen leesbare, bestanden met onze assembler-instructies te creëren.

We beginnen met het aanmaken van een *bronbestand* — de naam voor de tekstversie van een assembleerprogramma. We zullen nu zo'n bronbestand creëren, voor het programma dat we in hoofdstuk 3 hebben geschreven en toen Schrster hebben genoemd. Om uw geheugen op te frissen, laten we nog even de Debug-versie zien:

```
396F:0100 B402      MOV     AH,02
396F:0102 B22A      MOV     DL,2A
396F:0104 CD21      INT      21
396F:0106 CD20      INT      20
```

Gebruik uw editor om de volgende regels code in een bestand met de naam *SCHRSTER.ASM* te zetten (de uitbreiding *ASM* betekent dat het een bronbestand voor de assembler is). Net als bij Debug kunt u net zo goed kleine als hoofdletters gebruiken, maar wij blijven hoofdletters gebruiken om verwarring tussen het cijfer 1 (een) en de kleine letter l (el) te voorkomen:

```
.MODEL SMALL
.CODE

MOV     AH,2h
MOV     DL,2Ah
INT      21h
INT      20h
END
```

Dit is hetzelfde programma als we hebben gemaakt in hoofdstuk 3, maar het bevat een paar noodzakelijke wijzigingen en toevoegingen. Let even niet op de drie nieuwe regels van ons bronbestand, maar kijk naar de *h* achter elk hex-getal. Deze *h* betekent voor de assembler dat de getallen hexadecimaal zijn. Anders dan Debug, dat ervan uitgaat dat alle getallen hexadecimaal zijn, neemt de assembler aan dat alle getallen decimaal zijn. We vertellen hem dat het anders is door na elk hexadecimaal getal een *h* te zetten.

N.B. Een waarschuwing voor we verder gaan: de assembler kan in de war raken van getallen zoals ACh, die eruit zien als een naam of instructie. Om dit te voorkomen moet u altijd een nul tikken voor een hex-getal dat begint met een cijfer. Tik bijvoorbeeld 0ACh, en *niet* ACh.

Dit is een label

MOV DL,ACh

Dit is een getal

MOV DL,0ACh

De 0 betekent voor MASM
dat dit een getal is

Afb. 8-1. Zet een nul voor hexadecimale getallen die met een letter beginnen, anders behandelt de assembler het getal als een naam.

Kijk eens wat er gebeurt als we een programma met ACh inplaats van 0ACh assembleren. Dit is het programma:

```
.MODEL SMALL
.CODE

MOV     DL,ACh
INT     20h
END
```


Dit is de uitvoer:

```
C> MASM TEST:
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

a:TEST.ASM(2): error A2009: Symbol not defined: ACH

    3744 + 0 Bytes symbol space free

    0 Warning Errors
    1 Severe Errors

C>
```

Bepaald niet bemoedigend. Maar de assembler is tevreden als u ACh verandert in 0ACh.

We hebben gebruik gemaakt van versie 5.00 van de Macro-Assembler van Microsoft. Ten opzichte van de vorige versie (4.00) van deze assembler is er de mogelijkheid bijgekomen om in een verkorte vorm de diverse segmenten aan te geven. Dat is de vorm zoals in het voorbeeldprogrammaatje aangeduid. Het zou er in de source-code van 4.00 (dus niet-verkort) als volgt uitzien:

```
CODE_SEG      SEGMENT
               MOV     AH,2h
               MOV     DL,2Ah
               INT     21h
               INT     20h
CODE_SEG      ENDS
END
```

Segmenten worden dus in versie 4.00 aangegeven met de pseudo-ops SEGMENT en ENDS, terwijl de segmenten zelf benoemd worden. In versie 5.00 is het onder bepaalde omstandigheden voldoende om het MODEL aan te geven (*small*, *medium*, *large*, *huge*) en de segmenten aan het begin met .CODE voor een codesegment, .DATA voor een datasegment en .STACK voor een stacksegment. Voor de programma's die u normaal gesproken zult ontwikkelen, is het voorlopig voldoende ervan uit te gaan dat u werkt met MODEL SMALL (één codesegment en één datasegment) en dat u derhalve gebruik kunt maken van de verkorte vorm zoals in de eerste versie van de listing van ons voorbeeldprogrammaatje aangegeven. Wanneer u met versie 5.00 van de Macro-assembler werkt, kunt u echter ook voor versie 4.00 bedoelde source-code verwerken. In het restant van dit boek zal dan ook de notatie worden gebruikt van versie 4.00, zodat gebruikers van beide versies ermee uit de voeten kunnen.

Laten we nu teruggaan naar de eerste drie regels van ons tekstbestand (volgens de 4.00-versie). De drie nieuwe regels zijn alle drie *pseudo-ops*, (pseudo-operaties). Ze worden zo genoemd omdat ze, in plaats van instructies te genereren, de assembler alleen informatie verschaffen. De pseudo-op END markeert het einde van het bronbestand, zodat de assembler weet dat hij klaar is als hij een END ziet. Verderop zullen we zien dat END ook nog in andere opzichten nuttig is. Maar voorlopig laten

we een verdere bespreking daarvan of van de twee andere pseudo-ops voor wat ze zijn, en gaan we zien hoe de assembler moeten worden gebruikt.

8.2 Aanmaken van bronbestanden

Al hebt u de regels van SCHRSTER.ASM ingevoerd, we moeten nog een ding bekijken voor we ons programma echt kunnen assembleren. De assembler kan alleen bronbestanden gebruiken die standaard ASCII-tekens bevatten. Als u een tekstverwerker gebruikt, denk er dan aan dat niet alle tekstverwerkers bestanden schrijven die alleen de standaard ASCII-tekens bevatten. Wordstar is een van de boosdoeners; een andere is Microsoft Word. Gebruik voor deze beide tekstverwerkers de niet-document-, of ongeformatteerde, modus wanneer u uw bestanden bewaart.

Overtuig u ervan, voor u SCHRSTER.ASM probeert te assembleren, dat het nog in ASCII staat. Tik vanuit DOS:

```
C>TYPE SCHRSTER.ASM
```

U moet nu dezelfde tekst zien als u hebt ingevoerd. Als u vreemde tekens in uw programma ziet, moet u misschien een andere editor of tekstverwerker gebruiken om uw programma's in te voeren. U moet na de END-opdracht ook een lege regel zien. Laten we nu Schrster eens assembleren (vergeet niet de puntkomma aan het einde te tikken).

```
C> MASM SCHRSTER;
```

```
Microsoft (R) Macro Assembler Version 5.00
```

```
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.
```

```
51748 + 409596 Bytes symbol space free
```

```
0 Warning Errors
```

```
0 Severe Errors
```

```
C>
```

We zijn nog niet klaar. De assembler heeft nu een bestand gemaakt met de naam SCHRSTER.OBJ, dat u nu op uw schijf kunt aantreffen. Dit is een tussenbestand, een zogenaamd *werkbestand*. Het bestand bevat ons programma in machinetaal, met nog allerlei extra informatie die wordt gebruikt door een ander DOS-programma met de naam *linker*.

8.3 Linken

De linker is een koppelingsprogramma dat onze .OBJ-bestanden omzet in een .EXE-bestand. Daartoe moet u LINK.EXE in uw DOS-directory op de harde schijf hebben of van uw DOS-schijf kopiëren naar de schijf met uw bronbestand en de assembler. Link SCHRSTER.OBJ dan door te tikken:

C>LINK SCHRSTER;

Microsoft (R) Overlay Linker Version 3.60

Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

LINK : warning L4021: no stack segment

C>

In sommige versies van de linker wordt een fout gemeld, maar dat geeft niet. De linker ziet dan de waarschuwing als een fout, maar het is hier eigenlijk wat we willen. De linker waarschuwt ons dat er geen stapelsegment is, maar op dit moment hebben we er ook geen nodig. Als we nog meer over dit soort zaken hebben geleerd, zien we wel waarom een stapelsegment misschien nodig is.

We hebben nu ons .EXE-bestand, maar het is nog niet de laatste stap. We moeten nog een ding doen: een .COM-versie maken, hetzelfde als Debug voortbracht. Later zult u weer zien waarom ook deze stap nodig is. Laten we eerst maar een .COM-versie van Schrster maken.

Voor deze laatste stap hebben we het programma EXE2BIN van de supplementaire DOS-schijf nodig. Exe2bin zet, zoals de naam al aangeeft (2 uitgesproken als *to*, naar, i.p.v. *two*), een .EXE-bestand om in een .COM- of bin (binair)-bestand. Er is een verschil tussen .EXE- en .COM-bestanden, maar dat bespreken we pas later, dus laten we nu eerst maar het .COM-bestand maken. Tik:

C>EXE2BIN SCHRSTER SCHRSTER.COM

C>

De prompt zegt ons niet zo veel. Om te zien of Exe2bin gewerkt heeft, kunnen we alle Schrster-bestanden die we tot dusver hebben gemaakt, listen:

C>DIR SCHRSTER.*

Volume in drive C is FIXEDISK

Directory of C:\

SCHRSTER ASM	84	25-06-88	14:53
SCHRSTER OBJ	50	25-06-88	16:15
SCHRSTER EXE	520	25-06-88	16:17
SCHRSTER COM	8	25-06-88	16:18
4 File(s)		64512 bytes free	

C>

Dat zijn nogal wat bestanden. Het gewenste SCHRSTER.COM is er ook bij. Tik nu *schrster* om de .COM-versie te draaien en kijk of het programma goed werkt (het moet dus een sterretje op het scherm afbeelden). De precieze lengte die DOS voor de eerste drie bestanden opgeeft, kan wat verschillen.

Het resultaat valt misschien wat tegen, omdat we ogenschijnlijk nog even ver zijn als in hoofdstuk 3, maar dat is niet het geval: we zijn een stuk wijzer geworden. Het zal u allemaal veel duidelijker worden als we het weer over aanroepen (CALLs) hebben.

Het is u misschien opgevallen dat we ons geen moment druk hebben hoeven maken over waar ons programma in het geheugen kwam te staan, zoals over IP in Debug. Alle adressen zijn voor ons geregeld.

U zult deze eigenschap van de assembler snel leren waarderen: het programmeren wordt er veel gemakkelijker door. Denk bijvoorbeeld nog eens aan het laatste hoofdstuk, waarin we ruimte verspilden door ons hoofdprogramma op 100h en de procedure die we aanriepen op 200h te zetten. We zullen nog zien dat we de procedure met de assembler zonder enige tussenruimte onmiddellijk na het hoofdprogramma kunnen zetten. Maar laten we eerst eens zien hoe ons programma er voor Debug uit ziet.

8.4 Terug naar Debug

Laten we ons .COM-bestand eens met Debug bekijken en het disassembleren om te zien hoe Debug ons programma met de machinetaal van SCHRSTER.COM re-construeert:

```
C> DEBUG SCHRSTER.COM
```

```
-U
```

397F:0100 B402	MOV	AH,02
397F:0102 B22A	MOV	DL,2A
397F:0104 CD21	INT	21
397F:0106 CD20	INT	20

Precies hetzelfde als we in hoofdstuk 3 hadden. Dit is alles wat Debug in SCHRSTER.COM ziet. De END-instructie en de andere instructie over segmenten — CODE__SEG SEGMENT en CODE__SEG ENDS — zijn nergens meer te zien. Wat is ermee gebeurd?

Deze instructies staan niet in de uiteindelijke machinetaal-versie van het programma omdat het pseudo-ops zijn, en pseudo-ops zijn alleen voor administratieve doeleinden. De assembler verricht een hoop administratieve activiteiten die een paar extra regels nodig maken. We zullen de pseudo-ops nog goed leren gebruiken om ons werk te vereenvoudigen, en zien van welke invloed ze op ons programma zijn wanneer we in hoofdstuk 11 segmenten nader bekijken.

8.5 Commentaar

Omdat we niet meer rechtstreeks met Debug werken, kunnen we rustig meer aan ons programma toevoegen dat de assembler wel ziet maar niet doorgeeft aan de 8088. Misschien wel het belangrijkste daarvan is de mogelijkheid om er commentaar bij te zetten, dat van onschatbare waarde is voor het verduidelijken van het programma. In assembleerprogramma's zetten we commentaar na een puntkomma, die net zo werkt als een aanhalingsteken (') in BASIC. De assembler negeert alles wat er na een puntkomma op een regel staat, dus we kunnen eraan toevoegen wat we maar willen. Als we commentaar in ons programmaatje zetten:

CODE_SEG	SEGMENT	
MOV	AH,2h	;gebruik DOS-functie 2, uitvoer teken
MOV	DL,2Ah	;laad ASCII-code voor afdrukken '*'
INT	21h	;druk het af met INT
INT	20h	;en ga terug naar DOS
CODE_SEG	ENDS	
END		

zien we een hele verbetering — we kunnen dit programma begrijpen zonder terug te moeten denken en te onthouden wat elke regel ook weer betekent.

8.6 Labels

Ter afronding van dit hoofdstuk bekijken we nog een specifiek kenmerk van de assembler dat het programmeren vergemakkelijkt: labels.

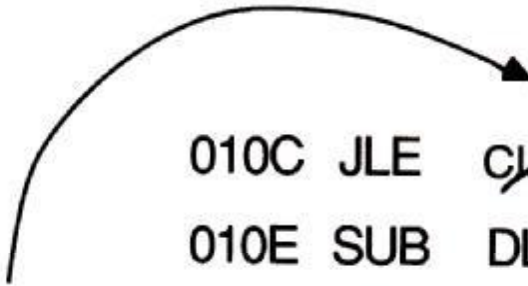
Als we tot dusver van het ene deel van het programma naar het andere wilden springen, moesten we het specifieke adres weten waar we naartoe sprongen. Bij het programmeren van alledag moet je bij het invoegen van nieuwe instructies de adressen in sprong-instructies wijzigen. De assembler regelt dit probleem met *labels* — namen die we geven aan de adressen van instructies of geheugenplaatsen. Een label komt in de plaats van een adres. Zodra de assembler een label ziet, vervangt hij het label door het juiste adres vóór hij het doorstuurt naar de 8088.

Labels mogen maximaal 31 tekens lang zijn en letters, cijfers en elk van de volgende tekens bevatten: vraagteken (?), punt (.), at-teken (@), underscore (_) of dollarteken (\$). Ze mogen niet beginnen met een cijfer (0 t/m 9) en een punt mag alleen als het eerste teken worden gebruikt.

Bekijk als praktisch voorbeeld eens ons programma van hoofdstuk 6, dat een hex-getal van twee cijfers inlas. Het bevat twee sprongen, JLE 0111 en JLE 011F. Dit is de oude versie:

3985:0100 B401	MOV	AH,01
3985:0102 CD21	INT	21
3985:0104 88C2	MOV	DL,AL
3985:0106 80EA30	SUB	DL,30
3985:0109 80FA09	CMP	DL,09
3985:010C 7E03	JLE	0111
3985:010E 80EA07	SUB	DL,07
3985:0111 B104	MOV	CL,04
3985:0113 D2E2	SHL	DL,CL
3985:0115 CD21	INT	21
3985:0117 2C30	SUB	AL,30
3985:0119 3C09	CMP	AL,09
3985:011B 7E02	JLE	011F
3985:011D 2C07	SUB	AL,07
3985:011F 00C2	ADD	DL,AL
3985:0121 CD20	INT	20

Het is bepaald niet duidelijk wat dit programma doet, en als het niet meer vers in uw geheugen zit, moet u zich misschien inspannen om het programma weer te begrijpen. Laten we er eens labels en commentaar bijzetten om de werking ervan te verduidelijken.



```

010C JLE CIJFER1
010E SUB DL
CIJFER1 : 0111 MOV CL
0113 SHL DL,1

```

Afb. 8-2. De assembler vervangt adressen door labels.

CODE_SEG	SEGMENT	
ASSUME	CS:CODE_SEG	;(wordt in hoofdstuk 11 uitgelegd)
MOV	AH,1h	;gebruik DOS-functie 1, invoer teken
INT	21h	;lees een teken en zet ASCII-code in AL
MOV	DL,AL	;verplaats ASCII-code naar DL
SUB	DL,30h	;trek 30h af voor omzetting naar 0-9
CMP	DL,9h	;was het een cijfer tussen 0 en 9?
JLE	CIJFER1	;ja, we hebben eerste cijfer (vier bits)
SUB	DL,7h	;nee, trek 7h af voor omzetting letter A-F
CIJFER1:		
MOV	CL,4h	;bereid voor op vermenigvuldigen met 16
SHL	DL,CL	;vermenigvuldig door verschuiven, wordt vier
		;hoogste bits
INT	21h	;haal volgende teken
SUB	AL,30h	;herhaal omzetting
CMP	AL,9h	;is het een cijfer 0-9?
JLE	CIJFER2	;ja, dan hebben we tweede cijfer
SUB	AL,7h	;nee, trek 7h af
CIJFER2:		
ADD	DL,AL	;ADD tweede cijfer
INT	20h	;en naar DOS
CODE_SEG	ENDS	
END		

De labels hier, CIJFER1 en CIJFER2, zijn van een type dat bekend staat als *NEAR*-labels omdat er een dubbelepunt achter wordt gezet bij de definitie. Het woord *NEAR* (nabij) heeft te maken met segmenten, waar we het in hoofdstuk 11 over zullen hebben, evenals over de pseudo-ops SEGMENT, ENDS en ASSUME. Als u het voorgaande programma assembleert en dan met Debug disassembleert, zult u zien dat CIJFER1 is vervangen door 0111h en CIJFER2 door 011Fh.

8.7 Samenvatting

Dat was me het hoofdstuk wel. Het is of we een nieuwe wereld zijn binnengestapt, en in zekere zin is dat ook zo. De assembler is veel gemakkelijker om mee te werken dan Debug, dus we kunnen nu echte programma's gaan schrijven omdat de assembler veel administratief werk voor ons doet.

Wat hebben we hier geleerd? Allereerst hoe we een bronbestand moeten aanmaken en het dan achtereenvolgens assembleren, linken en omzetten van een .OBJ- in een .EXE- en daarna in een .COM-bestand. We gebruikten daarvoor een eenvoudig programmaatje uit hoofdstuk 3. Het assembleerprogramma dat op die manier ontstond, bevatte enkele pseudo-ops, die we nog niet eerder hadden gezien, maar waarmee u wel vertrouwd zult raken naarmate u langer met de assembler werkt. Van nu af zullen de pseudo-ops SEGMENT, ENDS en END zelfs in al onze programma's komen te staan, ook al zien we pas in hoofdstuk 11 waarom dat eigenlijk moet.

Daarna hebben we over commentaar geleerd. U hebt u misschien afgevraagd hoe we het zonder commentaar hebben kunnen stellen. Van nu af doen we dat ook niet meer. Commentaar maakt programma's zo veel leesbaarder dat we er niet zuinig mee zullen zijn.

Ten slotte kwamen de labels, die onze programma's nog leesbaarder maken. In de rest van het boek blijven we al deze ideeën en methoden toepassen. We gaan nu door naar het volgende hoofdstuk en zullen daar zien hoe de assembler het gemakkelijker maakt om procedures te gebruiken.

9 Procedures en de assembler

- 9.1 De procedures van de assembler 102**
- 9.2 De procedures voor hex-uitvoer 104**
- 9.3 De beginselen van modulair ontwerpen 108**
- 9.4 Raamwerk van een programma 108**
- 9.5 Samenvatting 109**

Nu we de assembler hebben leren kennen, gaan we wat vertrouwder raken met het schrijven van programma's in assembleertaal. In dit hoofdstuk keren we terug naar het onderwerp van de procedures. U zult zien hoe we veel gemakkelijker we procedures kunnen schrijven met behulp van onze nijvere assembler. Daarna gaan we zelf enkele nuttige procedures opbouwen, die we dan gebruiken voor het ontwikkelen van ons Dskpatch-programma enkele hoofdstukken verderop.

We beginnen met twee procedures waarmee een byte in hexadecimaal wordt afgedrukt. We zullen daarbij nog allerlei andere pseudo-ops tegenkomen. Maar net als over SEGMENT, END en ENDS in het vorige hoofdstuk zullen we er pas in hoofdstuk 11 veel over zeggen, als we meer over segmenten vertellen.

9.1 De procedures van de assembler

Toen we voor de eerste keer procedures behandelden, lieten we een grote ruimte over tussen het hoofdprogramma en zijn procedures, zodat we plaats voor veranderingen zouden hebben zonder bang te hoeven zijn dat ons hoofdprogramma een procedure overlapte. Maar nu hebben we de assembler, en omdat die al het werk bij het toekennen van adressen aan instructies doet, hoeven we geen ruimte meer tussen procedures te laten bestaan. Met de assembler kunnen we na elke verandering het programma gewoon opnieuw assembleren.

In hoofdstuk 7 hebben we een programmaatje met een CALL gemaakt. Het enige wat dat programma deed, was de letters A tot en met J tonen, en het zag er als volgt uit:

3985:0100 B241	MOV	DL,41
3985:0102 B90A00	MOV	CX,000A
3985:0105 E8F800	CALL	0200
3985:0108 E2FB	LOOP	0105
3985:010A CD20	INT	20
3985:0200 B402	MOV	AH,02
3985:0202 CD21	INT	21
3985:0204 FEC2	INC	DL
3985:0206 C3	RET	

We gaan dit nu in een programma voor de assembler omzetten. Het zal zonder labels en commentaar moeilijk te lezen zijn, dus we zullen die verfraaiingen erbij zetten om ons programma veel leesbaarder te maken:

Listing 9-1. Het programma TOONA_J.ASM

```

CODE_SEG      SEGMENT
      ASSUME  CS:CODE_SEG
      ORG    100h           ;maak hier een .COM-bestand van (wordt
                           ;nog uitgelegd)
TOON_A_J      PROC    NEAR
      MOV     DL, 'A'       ;begin met het teken A
      MOV     CX,10        ;druk 10 tekens af, vanaf A
TOON_LUS:
      CALL    SCHRIJF_TEK   ;druk teken af, doorgaan naar volgende
      LOOP    TOON_LUS     ;ga tien tekens door
      INT     20h          ;terug naar DOS
TOON_A_J      ENDP

SCHRIJF_TEK   PROC    NEAR
      MOV     AH,2         ;functiecode voor uitvoer teken
      INT     21h          ;druk teken af dat al in DL staat
      INC     DL           ;ga door naar volgende teken in alfabet
      RET
SCHRIJF_TEK   ENDP

CODE_SEG      ENDS
      END      TOON_A_J

```

We hebben hier vier nieuwe pseudo-ops: ASSUME, ORG, PROC en ENDP. ASSUME heeft te maken met segmenten, en ORG met de manier waarop DOS programma's laadt; in hoofdstuk 11 komen we hier meer over te weten.

PROC en ENDP zijn pseudo-ops voor het definiëren van procedures. Zoals u ziet, staan zowel voor en na het hoofdprogramma als voor en na de procedure op 200h de bijbehorende pseudo-ops PROC en ENDP, die zelf weer worden omsloten door de pseudo-ops SEGMENT en ENDS (*End Segment*).

PROC geeft het begin van een procedure aan, ENDP het einde. Het label dat ervoor staat, is de naam die we geven aan de procedure die ze definiëren. Zo kunnen we in de hoofdprocedure, TOON_A_J, onze instructie CALL 200 vervangen door het leesbaardere CALL SCHRIJF_TEK. Geef gewoon de naam van de procedure en de assembler kent de adressen toe.

De pseudo-ops NEAR en FAR (over FAR verderop meer) verschaffen de assembler informatie over de manier waarop we segmenten gebruiken. De assembler gebruikt deze informatie steeds wanneer hij een CALL-instructie assembleert omdat er twee soorten CALL- en RET-instructies zijn: NEAR (nabij) en FAR (veraf). Een FAR-aanroep, die we hier niet zullen gebruiken, roept een procedure aan die in een ander segment zit. Een NEAR-aanroep roept echter een procedure aan die in hetzelfde segment staat.

In dit boek zullen we alleen programma's behandelen die in een enkel segment van 64K zitten, dus alle procedures zullen NEAR-procedures zijn. NEAR geeft de assembler te kennen dat de procedure in hetzelfde segment zit als de andere procedures die hij aanroept. Wanneer de assembler CALL SCHRIJF_TEK ziet, weet hij door de NEAR in SCHRIJF TEK PROC NEAR dat SCHRIJF_TEK in hetzelfde segment als TOON_A_J zit.

De assembler heeft deze informatie over segmenten nodig omdat er twee versies van

de CALL- en RET-instructie bestaan — een voor wanneer we niet van segment willen veranderen, en een voor wanneer we dat wel willen. Het is hier duidelijk dat onze twee procedures in hetzelfde segment staan, omdat we de beide procedures tussen twee bijeenhorende pseudo-ops voor het definiëren van segmenten hebben gezet: SEGMENT en ENDS. Verderop, als we ons programma in stukken verdelen die we in verschillende bronbestanden zetten, zal het gebruik van NEAR en FAR een belangrijkere rol gaan spelen.

Ten slotte moeten we, omdat we twee procedures hebben, de assembler vertellen welke hij als hoofdprocedure moet gebruiken — waar de 8088 moet beginnen met het uitvoeren van ons programma. Dit wordt geregeld door de pseudo-op END. Door END TOON__A__J in het programma te zetten, hebben we de assembler verteld dat TOON__A__J de hoofdprocedure is. Verderop zullen we zien dat de hoofdprocedure overal kan staan. Maar voorlopig houden we ons bezig met .COM-bestanden, en moeten we de hoofdprocedure vooraan in ons bronbestand zetten.

Als u het nog niet gedaan hebt, kunt u het programma invoeren in een bestand genaamd TOONA__J.ASM en de .COM-versie genereren volgens de stappen die in het vorige hoofdstuk hebt gezet.

```
MASM TOONAJ;  
LINK TOONAJ;  
EXE2BIN TOONAJ TOONAJ.COM
```

Probeer dan Toonaj eens uit. (Zorg wel dat u Exe2bin hebt gedraaid vóórdat u Toonaj draait. Anders draait u de .EXE-versie van Toonaj, die vast niet het resultaat geeft dat u verwacht.)

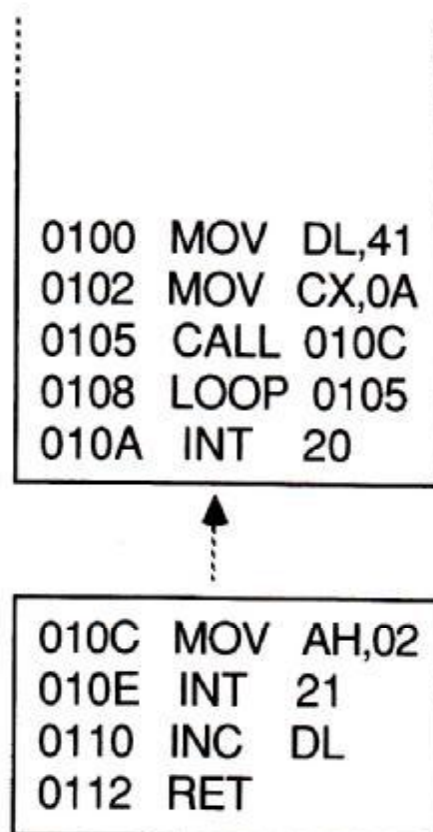
Als u tevreden bent met wat u ziet, moet u met Debug ons programma disassembleren en kijken hoe de assembler de twee procedures aan elkaar past. U zult nog weten dat we een bepaald bestand in Debug kunnen inlezen door de naam ervan op de commandoregel te tikken. Tik nu bijvoorbeeld *DEBUG TOONA__J.COM* en U; daarna ziet u:

3985:0100 B241	MOV	DL,41
3985:0102 B90A00	MOV	CX,000A
3985:0105 E80400	CALL	010C
3985:0108 E2FB	LOOP	0105
3985:010A CD20	INT	20
3985:010C B402	MOV	AH,02
3985:010E CD21	INT	21
3985:0110 FEC2	INC	DL
3985:0112 C3	RET	

Ons programma staat nu keurig achter elkaar, zonder enige ruimte tussen de twee procedures.

9.2 De procedures voor hex-uitvoer

We hebben al twee keer procedures gezien waarmee hex-getallen werden afgedrukt: een in hoofdstuk 5, waarin we leerden hoe we een getal in hex konden afdrukken, en een in hoofdstuk 7, toen we zagen hoe het programma kon worden vereenvoudigd



Afb. 9-1. MASM assembleert afzonderlijke procedures zonder tussenruimten.

met een procedure waarmee een hex-cijfer kon worden afgedrukt. Nu gaan we er nog een procedure aan toevoegen waarmee een teken wordt afgedrukt. Waarom? Noem het een vooruitziende blik.

Door een centrale procedure te gebruiken voor het schrijven van een teken naar het scherm, kunnen we de manier veranderen waarop deze procedure tekens naar het scherm schrijft zonder dat dat van invloed is op de rest van het programma. We zullen hem nog meermalen veranderen.

Voer het volgende programma in en noem het bestand VIDEO_IO.ASM:

Listing 9-2. Het nieuwe bestand VIDEO_IO.ASM

```

CODE_SEG      SEGMENT
    ASSUME    CS:CODE_SEG
    ORG       100h

TEST_SCHRIJF_HEX  PROC    NEAR
    MOV       DL,3Fh        ;test met 3Fh
    CALL      SCHRIJF_HEX
    INT       20h           ;terug naar DOS
TEST_SCHRIJF_HEX  ENDP

    PUBLIC    SCHRIJF_HEX

```


Listing 9-2. *vervolg*

```

;-----;
;Deze procedure zet de byte in het DL-register om in hex en schrijft ;
;de twee hex-cijfers naar de huidige cursorpositie. ;
; ;
; DL Naar hex om te zetten byte. ;
; ;
;Gebruikt: SCHRIJF_HEX_CIJFER ;
;-----;

SCHRIJF_HEX PROC NEAR ;ingangspunt
    PUSH CX ;bewaar in deze procedure gebruikte
            ;registers
    PUSH DX
    MOV DH,DL ;maak kopie van byte
    MOV CX,4 ;zet hoogste nibble in DL
    SHR DL,CL
    CALL SCHRIJF_HEX_CIJFER ;druk eerste hex-cijfer af
    MOV DL,DH ;zet laagste nibble in DL
    AND DL,0Fh ;verwijder hoogste nibble
    CALL SCHRIJF_HEX_CIJFER ;druk twee hex-cijfer af
    POP DX
    POP CX
    RET
SCHRIJF_HEX ENDP

PUBLIC SCHRIJF_HEX_CIJFER

;-----;
;Deze procedure zet de laagste 4 bits van DL om in een hex-cijfer en ;
;schrijft het naar het scherm. ;
; ;
; DL Laagste 4 bits bevatten in hex af te drukken getal. ;
; ;
; Gebruikt: SCHRIJF_TEK ;
;-----;

SCHRIJF_HEX_CIJFER PROC NEAR
    PUSH DX ;bewaar gebruikte registers
    CMP DL,10 ;is deze nibble <10?
    JAE HEX_LETTER ;nee, zet om in letter
    ADD DL,"0" ;ja, zet om in cijfer
    JMP Short SCHRIJF_CIJFER ;schrijf nu dit teken
HEX_LETTER:
    ADD DL,"A"-10 ;zet om in hex-letter
SCHRIJF_CIJFER:
    CALL SCHRIJF_TEK ;druk letter af op scherm
    POP DX ;zet oude waarde van AX terug
    RET
SCHRIJF_HEX_CIJFER ENDP
PUBLIC SCHRIJF_TEK

;-----;
; Deze procedure drukt een teken op het scherm af met behulp van de ;
; DOS-functie-aanroep. ;
; ;
; DL Op scherm af te drukken byte. ;
;-----;

```

Listing 9-2. *vervolg*

```

SCHRIJF_TEK      PROC      NEAR
    PUSH        AX
    MOV         AH,2                ;uitvoer teken aanroepen
    INT         21h                ;uitvoer teken in DL-register
    POP         AX                ;herstel oude waarde in AX
    RET                     ;en keer terug
SCHRIJF_TEK      ENDP

CODE_SEG        ENDS

                END      TEST_SCHRIJF_HEX

```

De DOS-functie voor het schrijven van tekens behandelt sommige tekens op een bepaalde manier. Zo leidt de DOS-functie voor het uitvoeren van 07 tot een pieptoon, zonder dat het teken voor 07, een klein ruitje, wordt afgedrukt. We zullen een nieuwe versie van SCHRIJF_TEK zien in deel 3, als we het hebben over de ROM-BIOS-routines binnenin uw IBM-PC. Voorlopig gebruiken we echter alleen de DOS-functie voor het afdrukken van tekens.

De nieuwe pseudo-op PUBLIC is erbij gezet voor toekomstig gebruik: in hoofdstuk 13, wanneer we meer over modulair ontwerpen leren. PUBLIC vertelt de assembler gewoon dat hij bepaalde informatie voor de linker moet genereren. Met behulp van de linker kunnen we bepaalde delen van ons programma, geassembleerd vanuit verschillende bronbestanden, tot één programma samenkoppelen. En PUBLIC betekent voor de assembler dat de procedure die wordt vermeld ná de pseudo-op PUBLIC algemeen beschikbaar moet worden gemaakt voor procedures in andere bestanden. Op dit moment bevat Video_io de drie procedures voor het schrijven van een byte als een hex-getal, en een kort programma om deze procedures te testen. We zullen bij het ontwikkelen van Dskpatch nog allerlei procedures aan het bestand toevoegen, en aan het eind van dit boek zal VIDEO_IO talrijke procedures voor algemeen gebruik bevatten.

De procedure TEST_SCHRIJF_HEX die we erbij hebben gezet, doet precies wat de naam zegt: hij test SCHRIJF_HEX, die op zijn beurt SCHRIJF_HEX_CIJFER en SCHRIJF_TEK gebruikt. Zodra we hebben gecontroleerd of deze procedures alle drie goed werken, halen we TEST_SCHRIJF_HEX uit VIDEO_IO.ASM. Maak de .COM-versie van Video_io aan, en gebruik Debug om SCHRIJF_HEX grondig te testen. Verander de 3Fh op geheugenplaats 101h in elk van de grenscondities die we in hoofdstuk 5 hebben geprobeerd en gebruik dan de G-opdracht om TEST_SCHRIJF_HEX te draaien.

We zullen veel eenvoudige testprogramma's gebruiken om nieuwe procedures te testen die we hebben geschreven. Op deze manier kunnen we een programma stukje bij beetje in elkaar zetten, in plaats van te proberen het in één keer te schrijven en te debuggen. Zo'n geleidelijke opbouw gaat veel sneller en is veel gemakkelijker, omdat we weten dat fouten alleen maar in de nieuwe code kunnen zitten.

9.3 De beginselen van modulair ontwerpen

Merk op dat we voor elke procedure in Video__io een blokje commentaar hebben gezet waarin de functie van elke procedure wordt beschreven. Belangrijker nog, dit commentaar geeft aan welke registers de procedure gebruikt om informatie heen en weer te sturen, en welke andere procedures hij gebruikt. Als een van de kenmerken van onze modulaire opbouw, stelt het commentaarblok ons in staat elke procedure te gebruiken door de beschrijving ervan te lezen. Je hoeft dan niet opnieuw te leren hoe de procedure zijn werk verricht. Dat maakt het ook vrij gemakkelijk om een procedure te herschrijven zonder ook procedures die erdoor worden aangeroepen te moeten herschrijven.

We hebben ook PUSH- en POP-instructies gebruikt om registers te bewaren en terug te zetten die we binnen elke procedure gebruiken. We zullen dat doen bij elke procedure die we schrijven, behalve bij onze testprocedures. Ook deze benadering is een onderdeel van de modulaire stijl die we zullen toepassen.

U zult nog wel weten dat we elk gebruikt register bewaren en herstellen zo dat we ons geen zorgen hoeven te maken over ingewikkelde interacties tussen procedure die elkaar bestrijden om het kleine aantal registers in de 8088 te kunnen gebruiken. Het staat elke procedure vrij om net zo veel registers te gebruiken als hij wil, *mits* hij ze vóór de RET-instructie weer in de oude staat herstelt. Het is een kleine prijs die moet worden betaald voor de extra eenvoud. Bovendien zou zonder het bewaren en herstellen van registers het herschrijven van procedures een klus zijn om gek van te worden. U zou er flink veel haar bij verliezen.

We proberen ook veel kleine procedures te gebruiken, in plaats van één grote. Ook dat maakt het programmeren veel gemakkelijker, hoewel we soms langere procedures zullen moeten schrijven wanneer het ontwerp bijzonder ingewikkeld wordt.

Deze ideeën en benadering worden in de volgende hoofdstukken allemaal nog uitgediept. In het volgende hoofdstuk zullen we bijvoorbeeld nog een procedure aan Video__io toevoegen: een procedure die een woord in het DX-register oppakt en het getal in decimaal op het scherm zet.

9.4 Raamwerk van een programma

Zoals we in dit en het vorige hoofdstuk al zagen, voegt de assembler aan elk programma dat we schrijven nog een aantal instructies van huishoudelijke aard toe. Met andere woorden, we moeten een paar pseudo-ops schrijven die de assembler wat elementaire dingen vertelt. Als houvast voor verderop laten we nu alvast het absolute minimum zien van wat u voor programma's nodig hebt:

```
CODE_SEG      SEGMENT
               ASSUME CS:CODE_SEG
               ORG    100h

Een_procedure  PROC    NEAR
               .
               .
               INT     20h
Een_procedure  ENDP
```

```
CODE_SEG      ENDS  
              Een_procedure  
END
```

In volgende hoofdstukken komen er nog enkele nieuwe pseudo-ops bij in dit programma, maar u zoals u het hier ziet, kunt u het gebruiken als uitgangspunt voor nieuwe programma's die u gaat schrijven. Of, beter nog, u kunt programma's en procedures in dit boek als uitgangspunt gebruiken.

9.5 Samenvatting

We schieten nu echt op. In dit hoofdstuk hebben we geleerd hoe we procedures in assembleertaal moeten schrijven. Van nu af aan zullen we de hele tijd procedures gebruiken, en met kleine procedures kunnen we onze programma's beter in de hand houden.

We zagen dat een procedure begint met een PROC-definitie en eindigt met aan pseudo-op ENDP. We hebben TOON_A_J herschreven om onze nieuwe kennis van procedures te testen, en toen ons nieuwe programma zodanig gewijzigd dat het een hex-getal schrijft — dit keer zonder extra procedure. Omdat procedures zo gemakkelijk mee te werken zijn, is er weinig reden om onze programma's niet in meerdere procedures op te delen. Sterker nog, we hebben gezien dat er talrijke redenen zijn om veel kleine procedures te gebruiken.

Aan het einde van dit hoofdstuk hebben we het even over modulair ontwerpen gehad, een methode die ons veel tijd en moeite zal besparen. Onze modulaire programma's zullen gemakkelijker te schrijven zijn, gemakkelijker te lezen en gemakkelijker door iemand anders te wijzigen dan programma's die zijn geschreven met de bekende spaghetti-logica: programma's met heel lange procedures en veel interacties.

We zijn nu klaar om een andere nuttige procedure op te bouwen. In hoofdstuk 11 leren we daarna over segmenten. En van dan af gaan we grotere programma's ontwikkelen, waarbij we de methode van het modulair ontwerpen ook echt zullen toepassen.

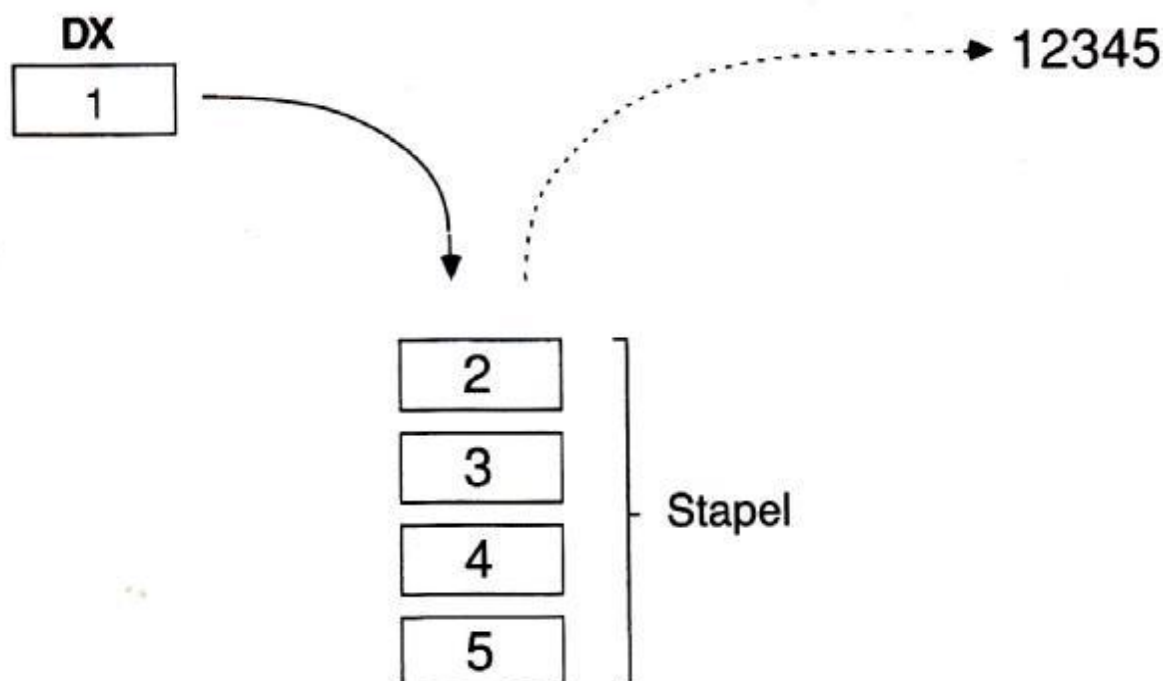
10 Afdrukken in decimaal

10.1	De conversie	112
10.2	Enkele trucs	114
10.3	Interne zaken	116
10.4	Samenvatting	117

We hadden u beloofd dat we een procedure zouden schrijven waarmee een woord in de decimale notatie zou worden afgedrukt. `SCHRIJF_DECIMAAL` gebruikt wat nieuwe trucs — om hier en daar wat microseconden en wat bytes te besparen. Misschien lijken zulke trucs nauwelijks de moeite waard. Maar als u ze onthoudt, zult u merken dat u ze kunt gebruiken om uw programma's korter en sneller te maken. Door die trucs leren we ook over twee nieuwe soorten logische bewerkingen, naast de `AND`-instructie die we in hoofdstuk 5 bespraken. Eerst bekijken we hoe een woord wordt omgezet in decimale cijfers.

10.1 De conversie

De sleutel bij het converteren of omzetten van een woord in decimale cijfers is deling. U zult nog weten dat de `DIV`-instructie zowel het hele getal (integer) als de rest als uitkomst geeft. Als je dus 12345 door 10 deelt, krijg je als uitkomst de integer 1234 met 5 als rest. In dit voorbeeld is 5 gewoon het meest rechtse cijfer. En als we weer door 10 delen, krijgen we het volgende cijfer daar links van. Een herhaalde deling door 10 haalt de cijfers er van rechts naar links af, en zet ze iedere keer in het restgetal.



Afb. 10-1. Door de cijfers op de stapel te zetten, komen ze daar in omgekeerde volgorde op te staan.

De cijfers komen er uiteraard in omgekeerde volgorde uit, maar bij het programmeren hebben we daar een oplossing voor. Weet u nog van de stapel? Die lijkt precies op een stapel borden: het eerste dat eraf gaat, is het laatste dat erop is gezet. Als we nu cijfers in plaats van borden nemen, en de cijfers bovenop elkaar zetten in de volg-

orde waarmee ze uit het restgetal komen, zitten we goed. Dan kunnen we de cijfers er in de juiste volgorde afhalen.

Het bovenste cijfer is het eerste cijfer van ons getal, en de andere cijfers staan eronder. Dus als we de resten bij de berekening steeds op de stapel zetten, en ze afdrukken wanneer we ze er weer afhalen, zullen de cijfers in de juiste volgorde staan.

Het volgende programma is de complete procedure voor het afdrukken van een getal in de decimale notatie. Zoals gezegd, schuilen er nog een paar trucs in deze procedure. Daar zullen we het zo over hebben, maar laten we eerst SCHRIJF_DECIMAAL eens proberen om te zien of hij werkt voor we ons druk maken over hoe hij werkt. Zet SCHRIJF_DECIMAAL in VIDEO_IO.ASM, samen met de procedures voor het schrijven van een byte in hex. Zorg wel dat SCHRIJF_DECIMAAL ná TEST_SCHRIJF_HEX komt te staan, dat we zullen vervangen door TEST_SCHRIJF_DECIMAAL, Om werk te besparen, maakt SCHRIJF_DECIMAAL gebruik van SCHRIJF_HEX_CIJFER om een nibble (vier bits) om te zetten in een cijfer.

Listing 10-1. Toevoegen aan VIDEO_IO.ASM

```

PUBLIC SCHRIJF_DECIMAAL
;-----;
; Deze procedure schrijft een 16-bits getal zonder teken in decimale ;
; notatie. ;
; ;
; DX      N : 16-bits getal zonder teken. ;
; ;
; Gebruikt:  SCHRIJF_HEX_CIJFER ;
;-----;
SCHRIJF_DECIMAAL PROC NEAR
    PUSH    AX                ;bewaar hier gebruikte registers
    PUSH    CX
    PUSH    DX
    PUSH    SI
    MOV     AX,DX              ;zet het getal in AX
    MOV     SI,10              ;deelt door 10 met gebruik van SI
    XOR     CX,CX              ;aantal cijfers op stapel
NIET_NUL:
    XOR     DX,DX              ;maakt hoogste woord van N gelijk aan 0
    DIV     SI                 ;bereken N/10 en (N mod 10)
    PUSH    DX                 ;zet een cijfer op de stapel
    INC     CX                 ;nog een cijfer erbij
    OR      AX,AX              ;is N al 0?
    JNE     NIET_NUL           ;nee, doorgaan
SCHRIJF_CIJFER_LUS:
    POP     DX                 ;haal cijfers in omgekeerde volgorde op
    CALL    SCHRIJF_HEX_CIJFER
    LOOP    SCHRIJF_CIJFER_LUS
EINDE_DECIMAAL:
    POP     SI
    POP     DX
    POP     CX
    POP     AX
    RET
SCHRIJF_DECIMAAL ENDP

```


Merk op dat we een nieuw register, SI (*Source Index*, bronindex), hebben gebruikt. Verderop zien we wel waarom het die naam heeft, en zullen we ook kennis maken met zijn broertje DI (*Destination Index*, bestemmingsindex). Beide registers worden op een bepaalde manier gebruikt, maar ze kunnen ook als algemene registers functioneren. Omdat SCHRIJF_DECIMAAL vier algemene registers nodig heeft, hebben we SI gebruikt, al hadden we ook BX kunnen nemen, gewoon om te laten zien dat SI (en DI) zo nodig als algemeen register kan dienen.

Voor we onze nieuwe procedure uitproberen, moeten we nog twee veranderingen in VIDEO_IO.ASM aanbrengen. Allereerst moeten we de procedure TEST_SCHRIJF_HEX eruit halen, en deze procedure ervoor in de plaats zetten:

Listing 10-2. Vervang TEST_SCHRIJF_HEX in VIDEO_IO.ASM door deze procedure

```
TEST_SCHRIJF_DECIMAAL  PROC    NEAR
                        MOV     DX,12345
                        CALL    SCHRIJF_DECIMAAL
                        INT 20h                ;terug naar DOS
TEST_SCHRIJF_DECIMAAL  ENDP
```

Deze procedure test SCHRIJF_DECIMAAL met het getal 12345 (dat de assembler omzet in het woord 3039h).

Ten tweede moeten we de END-opdracht aan het einde van VIDEO_IO.ASM veranderen in END TEST_SCHRIJF_DECIMAAL omdat TEST_SCHRIJF_DECIMAAL nu onze hoofdprocedure is.

Breng deze veranderingen aan en probeer VIDEO.IO uit. Zet het om in zijn .COM-versie en kijk of het werkt. Zo niet, kijk dan in uw bronbestand of er fouten in staan. Bent u avontuurlijk van aard, probeer uw fout dan met Debug op te sporen. Daar is Debug tenslotte voor.

10.2 Enkele trucs

In SCHRIJF_DECIMAAL zitten een paar slimigheidjes verborgen die ook zijn toegepast door de mensen die de ROM BIOS-procedures hebben geschreven waarmee we in hoofdstuk 17 zullen kennismaken. Het eerste is een efficiënte instructie om een register op nul te zetten. Ze is niet veel efficiënter dan MOV AX,0, en misschien niet de moeite waard, maar het is wel het soort trucs dat de mensen van het vak toepassen, dus hier is hij. De instructie:

```
XOR    DX,DX
```

zet het DX-register op nul. Hoe dan? Om dat te begrijpen, moeten we de logische bewerking genaamd XOR (*eXclusive OR*, exclusieve 'of') leren kennen.

XOR lijkt op de gewone OR (waarover straks) maar het XOR-en van twee keer waar

XOR	0	1
0	0	1
1	1	0

geeft waar als er maar één bit waar is, niet als beide waar zijn. Als we dus een XOR op een getal zelf toepassen, is het resultaat nul:

```

      1 0 1 1 0 1 0 1
XOR 1 0 1 1 0 1 0 1
-----
      0 0 0 0 0 0 0 0

```

Dat is de kneep. We zullen XOR in dit boek verder niet meer nodig hebben, maar we dachten dat het u het wel interessant zou vinden om te weten.

Even tussen haakjes: u zult nog een andere truc zien die veel anderen gebruiken om een register op nul te zetten. In plaats van XOR hadden we

```
SUB    DX,DX
```

kunnen gebruiken om het DX-register op nul te zetten.

Nu de andere truc. Die is haast net zo slim en gebruikt een broertje van de exclusieve OR — de gewone OR.

We willen controleren of het AX-register op nul staat. Daartoe zouden we de instructie `CMP AX,0` kunnen gebruiken. Maar nee, we passen liever een truc toe. Dat is leuker en ook een beetje efficiënter. Dus we schrijven `OR AX,AX`, gevolgd door een voorwaardelijke sprong `JNE` (*Jump if Not Equal*, spring indien niet gelijk). (We hadden ook `JNZ`, *Jump on Not Zero*, spring indien niet nul, kunnen gebruiken.) Net als alle andere rekenkundige instructies stelt de OR-instructie de vlaggen in, ook de nulvlag. OR is, net als AND, een logische bewerking. Maar het resultaat is hier waar als het ene óf (OR) het andere bit waar is (of beide):

```

OR 0 1
   0 1
   1 1

```

Als we nu een getal nemen en met zichzelf OR-en, krijgen we het oorspronkelijke getal weer:

```

      1 0 1 1 0 1 0 1
OR 1 0 1 1 0 1 0 1
-----
      1 0 1 1 0 1 0 1

```

De OR-instructie is ook handig wanneer we maar één bit in een byte op 1 willen zetten. Je kunt bijvoorbeeld bit 3 in het getal van zoëven zetten:

```

      1 0 1 1 0 1 0 1
OR 0 0 0 0 1 0 0 0
-----
      1 0 1 1 1 1 0 1

```

We krijgen nog meer dan dit soort trucs in dit boek om mee te spelen, maar deze twee zijn de enige die alleen voor de lol gebruikt kunnen worden.

10.3 Interne zaken

Als u wilt zien hoe SCHRIJF_DECIMAAL zijn werk doet, moet u de listing bestuderen: we zullen het hier verder niet over de details hebben. We moeten echter nog wel op enkele dingen wijzen.

Allereerst moet u weten dat het CX-register gebruikt wordt om te tellen hoeveel cijfers we op de stapel hebben gezet, zodat we weten hoeveel we eraf moeten halen. Het CX-register is een erg voor de hand liggende keuze omdat we met de LOOP-instructie een lus kunnen opzetten en het CX-register gebruiken om de herhalingsteller op te slaan. De lus waarmee de cijfers worden uitgevoerd (SCHRIJF_CIJFER_LUS), stelt op deze manier haast niets voor omdat de LOOP-instructie het CX-register rechtstreeks gebruikt. We zullen CX nog vaak gebruiken wanneer we een telling moeten bewaren.

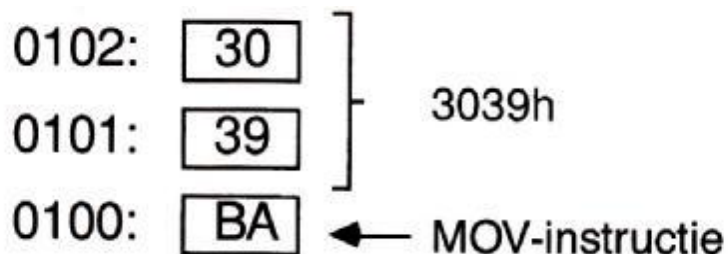
Zorg ook dat u hier goed de grenscondities test. De grensconditie bij 0 is geen probleem, zoals u zelf kunt nagaan. De andere grensconditie is 65535, of FFFFh, die u gemakkelijk met Debug kunt controleren. Laad gewoon VIDEO_IO.COM in Debug door *DEBUG VIDEO_IO.COM* te tikken en wijzig de 12345 (3039h) op 101h in 65535 (FFFFh). (SCHRIJF_DECIMAAL werkt met getallen zonder teken. Probeer eens een versie te maken die getallen met teken afdrukt.)

U hebt misschien een moeilijkheid bespeurd, in verband met de 8088, niet met ons programma. Debug werkt voornamelijk met bytes (althans de E-opdracht), maar we willen een woord veranderen. Daar moeten we voorzichtig mee zijn omdat de 8088 de bytes in omgekeerde volgorde bewaart. De MOV-instructie ziet er gedissassembled zo uit:

```
3985:0100 BA3930      MOV     DX,3039
```

U ziet aan de *BA3930* in deze regel dat de byte op 101h 39h is, en die op 102h 30h (BA is de MOV-instructie). De twee bytes zijn de twee bytes van 3039h, maar zo te zien in omgekeerde volgorde. Verwarrend? In feite zal het u heel logisch voorkomen als we het even hebben uitgelegd.

MOV DX,3039



Afb. 10-2. De 8088 slaat getallen met eerste de lage byte in het geheugen op.

Een *woord* bestaat uit twee delen, de lage en de hoge byte. De lage byte is de minst belangrijke byte (39h in 3039h) terwijl de hoge byte het andere deel is (30h). Het is daarom zinnig om de laagste byte op het laagste adres in het geheugen te zetten. (Sommige computers keren deze twee bytes weer om, en als u met meerdere computers werkt, kan dat wat verwarring geven.)

Probeer eens verschillende getallen voor het woord dat begint op 101h, en u zult zien hoe ze worden opgeslagen. Gebruik `TEST__SCHRIJF__DECIMAAL` om te zien of het allemaal klopt, of disassembleer de eerste instructie.

10.4 Samenvatting

We hebben weer enkele instructies aan ons repertoire toegevoegd, en voor de lol wat trucjes besproken. We hebben ook over twee andere registers, SI en DI geleerd, die we als algemene registers kunnen gebruiken. In volgende hoofdstukken zullen we zien dat ze ook nog op andere manieren te gebruiken zijn.

We hebben geleerd over de logische instructie XOR en OR, waarmee we met afzonderlijke bits in twee bytes of woorden kunnen werken. En in onze `SCHRIJF__DECIMAAL`-procedure gebruikten we de instructie `XOR DX,DX` als een truc om het DX-register op nul te zetten. We gebruikten `OR AX,AX` als een slim equivalent van `CMP AX,0` om het AX-register te testen en te zien of het op nul staat. Ten slotte hebben we geleerd hoe de 8088 een woord in het geheugen opslaat door de grenscondities van onze nieuwe procedure `SCHRIJF__DECIMAAL` na te gaan. We hebben er nu, aan het einde van dit hoofdstuk, nog een procedure voor algemene doeleinden bij, genaamd `SCHRIJF__DECIMAAL`, die we in de toekomst voor onze eigen programma's zullen kunnen gebruiken.

Haal nu even diep adem. We hebben een paar *andere* hoofdstukken voor u in petto. In hoofdstuk 11 worden segmenten in detail besproken. Segmenten vormen misschien wel het meest ingewikkelde onderdeel van de 8088-microprocessor, dus het zou best een nogal zwaar hoofdstuk kunnen zijn. Maar we moeten het onderwerp nu eenmaal bespreken in verband met de hoofdstukken daarna.

Daarna brengen we een kleine koerswijziging aan en komen we op ons spoor terug door te leren over wat we willen doen met ons programma `Dskpatch`. We gaan schrijven wat nader bekijken, en leren over sectoren, sporen en meer van die dingen.

Van dan af kunnen we een eenvoudige koers volgen met voorlopige versies van `Dskpatch` als richtlijn. Onderweg krijgt u een kans om te zien hoe grote programma's moeten worden ontwikkeld. Programmeurs schrijven niet een heel programma en halen dan de fouten eruit. Ze schrijven er delen van en proberen elk deel uit voor ze verder gaan — programmeren is op die manier veel gemakkelijker. We hebben deze benadering tot op zekere hoogte al aangehouden bij het schrijven en testen van `SCHRIJF__HEX` en `SCHRIJF__DECIMAAL`, waarvoor de testprogramma's heel eenvoudig waren. De testprogramma's zullen van nu af ingewikkelder zijn, maar ook interessanter.

11 Segmenten

- 11.1 Het geheugen van de 8088 in stukken verdelen 120**
- 11.2 Pseudo-ops voor segmenten 126**
- 11.3 De pseudo-op ASSUME 127**
- 11.4 NEAR- en FAR-aanroepen 128**
- 11.5 Meer over de INT-instructie 130**
- 11.6 Interrupt-vectoren 131**
- 11.7 Samenvatting 132**

In de vorige hoofdstukken kwamen we verscheidene pseudo-ops tegen die met segmenten werken. Nu is de tijd gekomen om te kijken naar de segmenten zelf en de manier waarop de 8088 een hele megabyte (1.048.576 bytes) aan geheugen adresseert. Daarna zullen we begrijpen waarom segmenten hun eigen pseudo-ops in de assembler nodig hebben, en in volgende hoofdstukken zullen we verschillende segmenten leren gebruiken (tot dusver hebben we er maar één gebruikt). Daarna, in hoofdstuk 13, wanneer we over modulaire opbouw leren, zullen we zien hoe segmenten kunnen worden samengebracht tot een .COM-bestand.

Laten we beginnen op het niveau van de 8088 en leren hoe die de 20-bits adressen samenstelt die nodig zijn voor een hele megabyte geheugen.

11.1 Het geheugen van de 8088 in stukken verdelen

Segmenten vormen ongeveer het enige deel van de 8088 dat we nog niet hebben besproken, en voor de meeste mensen zijn ze misschien wel het meest verwarrende onderdeel van de microprocessor. Sterker nog, in het computervak worden ze een *kludge* (zootje ongeregeld) genoemd, om aan te geven dat ze een noodoplossing van een probleem vormen.

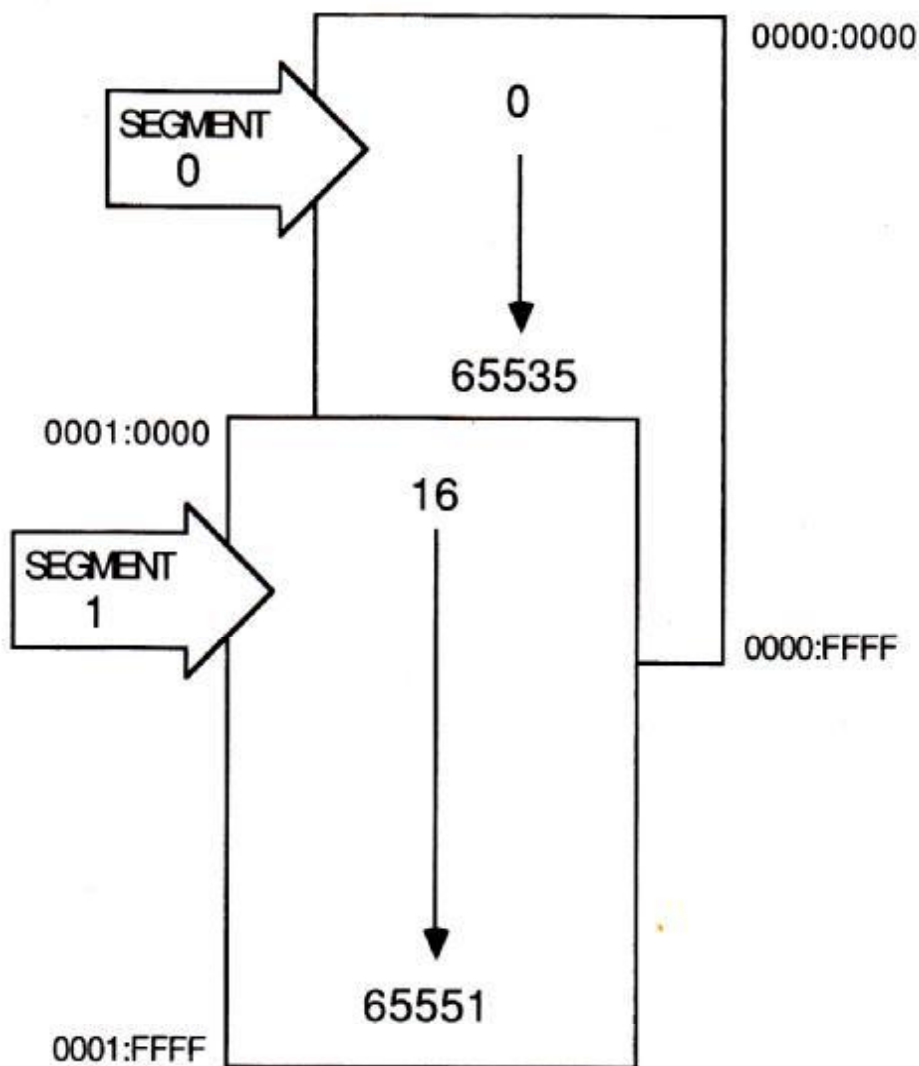
Het probleem is in dit geval om meer dan 64K geheugen te kunnen aanspreken — wat bij een woord de limiet is, omdat 65535 het grootste getal is dat een enkel woord kan bevatten. Intel, het bedrijf dat de 8088 heeft ontworpen, gebruikte segmenten en segmentregisters om dit probleem op te lossen, maar maakte de 8088 daardoor wel verwarrender.

Tot dusver hebben we ons nog niet met het probleem zelf bezig gehouden. Vanaf dat we Debug in hoofdstuk 2 leerden kennen, hebben we het IP-register gebruikt om het adres vast te houden van de volgende instructie die de 8088 moest uitvoeren. Misschien weet u nog dat we toen zeiden dat het adres in feite bestaat uit zowel het CS- als het IP-register. Maar we zeiden er niet bij hoe het precies zat. Dat gaan we nu bekijken.

Hoewel het volledige adres door twee registers wordt gevormd, maakt de 8088 geen getal van twee woorden aan voor het adres. Als u CS:IP als een 32-bits getal zou nemen (twee 16-bits getallen naast elkaar), zou de 8088 viermiljard bytes kunnen adresseren — veel meer dan de een miljoen die het in feite kan aanspreken. De methode van de 8088 is iets ingewikkelder. Het CS-register verschaft het *startadres* voor het codesegment, waarbij een segment uit 64K geheugen bestaat. Het werkt als volgt: Zoals u in afb. 11-1 kunt zien, verdeelt de 8088 geheugen in verscheidene elkaar overlappende segmenten, waarbij na elke 16 bytes een nieuw segment begint. Het eerste segment (segment 0) begint op geheugenplaats 0; het tweede (segment 1) begint op 10h (16); het derde op 20h (32), enz..

Het feitelijke adres is gewoon $CS * 16 + IP$. Als er bijvoorbeeld in het CS-register 3FA8 staat en in IP D017, dan is het absolute adres:

CS * 16 :	0	0	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0
IP :					1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	1
	0	1	0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	1	1	1



Afb. 11-1. Elkaar overlappende segmenten beginnen op elke 16 bytes, en zijn 65536 bytes lang.

We hebben met 16 vermenigvuldigd door gewoon CS vier bits naar links te schuiven, en er aan de rechterkant nullen bij te zetten.

Dit lijkt misschien een vreemde manier om meer dan 64K geheugen te adresseren, en dat is het ook — maar het werkt wel. We zullen zo zien hoe goed het wel werkt.

$$\begin{array}{r}
 \leftarrow \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline \end{array} & \leftarrow \text{Segment (CS)} \\
 + & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array} & \leftarrow \text{Offset (IP)} \\
 \hline
 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1
 \end{array}$$

Afb. 11-2. Het absolute adres van CS:IP is $CS * 16 + IP$.

De 8088 heeft eigenlijk vier segmentregisters: CS (CodeSegment), DS (DataSegment), SS (StapelSegment) en ES (Extra Segment). Het CS-register dat we hebben bekeken, wordt door de 8088 gebruikt voor het segment waarin de volgende instructie is opgeslagen. Op vrijwel dezelfde manier is DS het segment waarin de 8088 gegevens (data) opzoekt, en SS de plaats waar de 8088 zijn stapel heeft.

Voor we verder gaan, bekijken we even een kort programmaatje, heel anders dan de programma's die we tot dusver hebben gezien, dat gebruik maakt van twee verschillende segmenten. Voer dit programma in en noem het TEST_SEG.ASM.

Listing 11-1. Het programma TEST_SEG.ASM

```
CODE_SEG      SEGMENT
                ASSUME CS:CODE_SEG
TEST_SEGMENT  PROC    NEAR
                MOV     AH,4Ch                ;vraag functie voor terug naar DOS
                INT     21h                  ;keer terug naar DOS
TEST_SEGMENT  ENDP
CODE_SEG      ENDS

STAPEL_SEGMENT SEGMENT STACK
                DB      10 DUP ("Stapel ")    ;twee spaties na stapel
STAPEL_SEGMENT ENDS

                END      TEST_SEGMENT
```

Assembleer en link Test_seg nu, maar maak er geen .COM-bestand voor. Het resultaat zal TEST_SEG.EXE zijn, dat er iets anders uitziet dan een .COM-bestand.

N.B. We zullen een andere methode voor het beëindigen van .EXE-bestanden moeten toepassen. Bij .COM-bestanden werkt INT 20h uitstekend, maar absoluut niet bij .EXE-bestanden omdat de segmenten heel anders zijn ingedeeld, zoals we in dit hoofdstuk zullen zien; verderop gaan we dieper op het verschil in.

Toen we Debug voor een .COM-bestand gebruikten, gaf Debug alle segmentregisters dezelfde inhoud, en begon het programma op een *offset* (afstand) van 100h van het begin van dit segment. De eerste 256 bytes (100h) worden gebruikt om allerlei informatie op te slaan die niet echt van belang voor ons is, maar we zullen een deel van dit gebied straks even bekijken.

Probeer nu TEST_SEG.EXE in Debug te laden om te zien wat er met segmenten in een .EXE-bestand gebeurt:

C>DEBUG TEST_SEG.EXE

-R

```
AX=0000  BX=0000  CX=006A  DX=0000  SP=0050  BP=0000  SI=0000  DI=0000
DS=3985  ES=3985  SS=3996  CS=3995  IP=0000  NV UP DI PL NZ NA PO NC
3995:0000 CD20                INT     20
```

De waarden van de registers SS en CS verschillen van die van DS en ES.

In ons programma hebben we twee segmenten gedefinieerd. Het STAPEL_SEGMENT is waar we de stapel zetten (vandaar het woord *STACK* (stapel) na het woord

SEGMENT). We hebben de stapel als 80 bytes lang gedefinieerd: de instructie DB 10 DUP ('Stapel ') vertelt de assembler dat hij de string tussen aanhalingstekens in bytes moet omzetten, en de string dan tien keer in het geheugen moet herhalen. DB (*Define Byte*) zegt de assembler dat we geheugenbytes definiëren. We zetten hier tien herhalingen van de ASCII-code voor *Stapel* met twee spaties op de stapel. De code hiervoor is 53 74 61 70 65 6C 20 20, dus als we het stapelsegment bekijken, moeten we deze getallen tien keer herhaald zien. Vraag Debug dit geheugengebied te dumpen met de volgende opdracht, die Debug vertelt het geheugen vanaf offset 0 binnen het stapelsegment te dumpen:

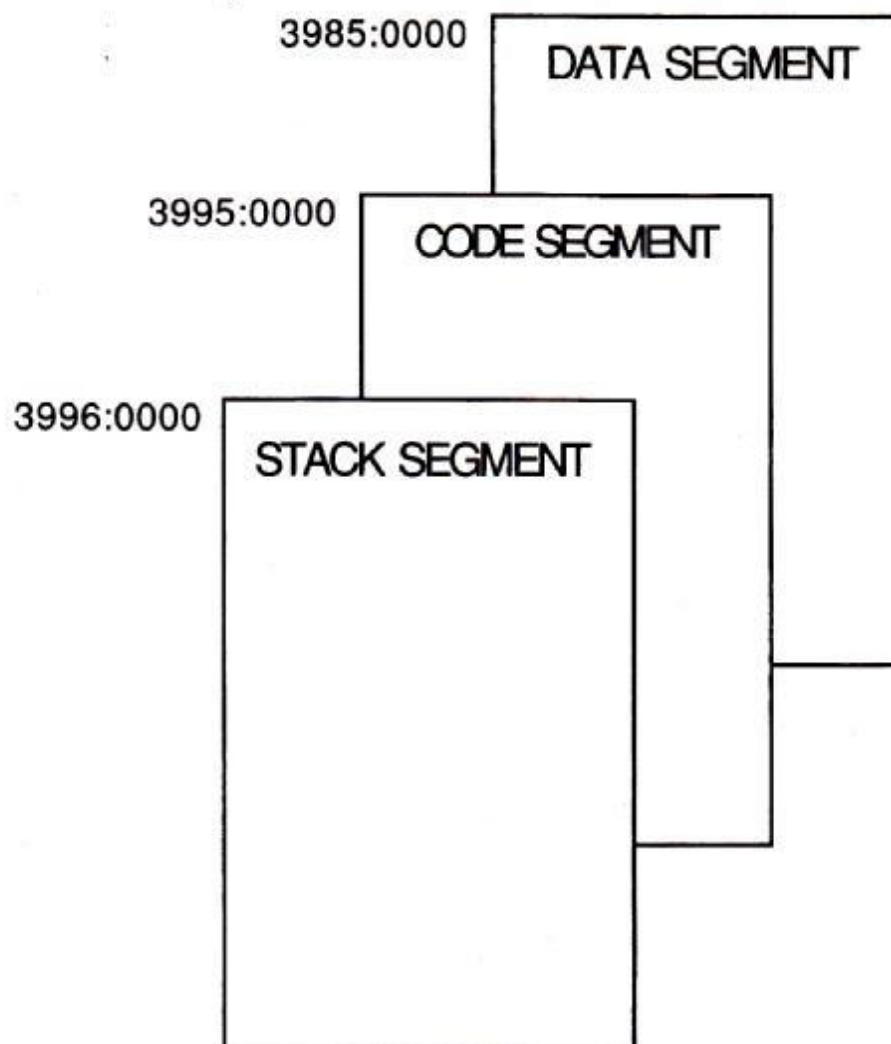
```
-D SS:0
3996:0000  53 74 61 70 65 6C 20 20-53 74 61 70 65 6C 20 20  Stapel Stapel
3996:0010  53 74 61 70 65 6C 20 20-53 74 61 70 65 6C 20 20  Stapel Stapel
3996:0020  53 74 61 70 65 6C 20 20-53 74 61 70 65 6C 20 20  Stapel Stapel
3996:0030  53 74 61 70 65 6C 20 20-53 74 61 70 65 6C 20 20  Stapel Stapel
3996:0040  53 74 61 70 65 6C 20 20-53 74 61 70 65 6C 00 00  Stapel Stapel..
3996:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  .....
.
.
.
↑  ↑
SS SP
```

Het adres van de top van de stapel wordt gegeven met SS:SP. SP is de stapelwijzer (*Stack Pointer*), net als IP en CS voor code, en is een offset binnen het huidige stapelsegment.

Eigenlijk is 'top van de stapel' een verkeerde benaming, omdat de stapel van boven naar beneden in het geheugen groeit. De top van de stapel is dus eigenlijk de onderkant van de stapel in het geheugen, en nieuwe ingangen van de stapel worden steeds lager in het geheugen gezet. SP is hier 50h, wat 80 decimaal is, omdat we een stapelgebied van 80 bytes lang hebben gedefinieerd. We hebben tot dusver nog niets op de stapel gezet, dus de top-van-de-stapel is nog steeds het begin van het geheugen dat we voor de stapel hebben gereserveerd: 50h.

Nu u weet hoe u de stapel kunt vinden, zou u kunnen kijken hoe hij bij de programma's in vorige hoofdstukken verandert. Maar we gaan hier door met het voorbeeld dat al in Debug zit.

Merk op dat het stapelsegment (SS) het segmentnummer 3996 heeft (bij u is het waarschijnlijk een ander nummer), terwijl ons codesegment (CS) op 3995 staat — één minder dan SS, ofwel precies 16 bytes lager in het geheugen. Dat betekent dat als we het geheugen disassembleren vanaf CS:0, we ons programma zullen zien gevolgd door 12 bytes die gelijk aan nul zijn (de MOV AH,4C en INT 21h beslaan elk twee bytes), en daarna zien we de bytes van het stapelsegment. We zien ook de gedisassembleerde gegevens voor *Stapel* gevolgd door twee spaties:



Afb. 11-3. Geheugenoverzicht voor TEST_SEG.EXE.

-U CS:0	
3995:0000 B44C	MOV AH,4C
3995:0002 CD21	INT 21
3995:0004 0000	ADD [BX+SI],AL
3995:0006 0000	ADD [BX+SI],AL
3995:0008 0000	ADD [BX+SI],AL
3995:000A 0000	ADD [BX+SI],AL
3995:000C 0000	ADD [BX+SI],AL
3995:000E 0000	ADD [BX+SI],AL
3995:0010 53	PUSH BX
3995:0011 7461	JZ 0074
3995:0013 7065	JO 007A
3995:0015 6C	DB 6C
3995:0016 2020	AND [BX+SI],AH
3995:0018 53	PUSH BX
3995:0019 7461	JZ 007C
3995:001B 7065	JO 0082

```

3995:001D 6C          DB      6C
3995:001E 2020        AND     [BX+SI],AH

```

Net zoals we verwachtten, staat het getal 53 — de ASCII-code voor S, de eerste letter van ons stapelgebied — op offset 10h (16) binnen ons codesegment.

Bij het bekijken van de registerafbeelding is het u misschien opgevallen dat de registers ES en DS het getal 3985h bevatten, 10h minder dan het begin van het programma in segment 3995. Als we dat met 16 vermenigvuldigen om het aantal bytes te krijgen, zien we dat er 100h (of 256) bytes vóór het begin van het programma staan. Dat is hetzelfde gebied als aan het begin van een .COM-bestand wordt gezet. Dit *kladgebied* van 256 bytes aan het begin van programma's bevat de tekens die we tikken na de naam van het programma. Bijvoorbeeld:

C>DEBUG TEST_SEG.EXE En nu wat tekens die we in de geheugendump zullen zien

```

-D DS:80
3985:0080  38 20 45 6E 20 6E 75 20-77 61 74 20 74 65 6B 65   8 En nu wat teke
3985:0090  6E 73 20 64 69 65 20 77-65 20 69 6E 20 64 65 20   ns die we in de
3985:00A0  67 65 68 65 75 67 65 6E-64 75 6D 70 20 7A 75 6C   geheugendump zul
3985:00B0  6C 65 6E 20 7A 69 65 6E-20 0D 7A 75 6C 6C 65 6E   len zien .zullen
3985:00C0  20 7A 69 65 6E 20 0D 00-00 00 00 00 00 00 00     zien .....
3985:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00     .....
3985:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00     .....
3985:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00     .....

```

De eerste byte geeft aan dat we 38h (ofwel 56) tekens hebben getikt, met inbegrip van de eerste spatie na TEST__SEG.EXE. We zullen deze informatie in dit boek niet gebruiken, maar het laat wel zien waarom u zo'n groot kladgebied nodig zou kunnen hebben.

N.B. Het 'kladgebied' heet eigenlijk PSP (Program Segment Prefix) en bevat informatie die door DOS wordt gebruikt. Met andere woorden, u moet er niet van uitgaan dat u van dit gebied gebruik kunt maken.

Het kladgebied bevat ook informatie die DOS gebruikt wanneer we een programma verlaten met de instructies INT 20h of INT 21h, functie 4Ch. Maar om redenen die geheel onduidelijk zijn, verwacht de instructie INT 20h dat het CS-register naar het begin van dit kladgebied wijst, wat het ook doet bij een .COM-programma, maar *niet* bij een .EXE-programma. Dit is een historische kwestie. Bovendien is het zo dat de functie voor het verlaten van het programma (INT 21h, functie 4Ch) bij de introductie van versie 2.00 aan DOS is toegevoegd.

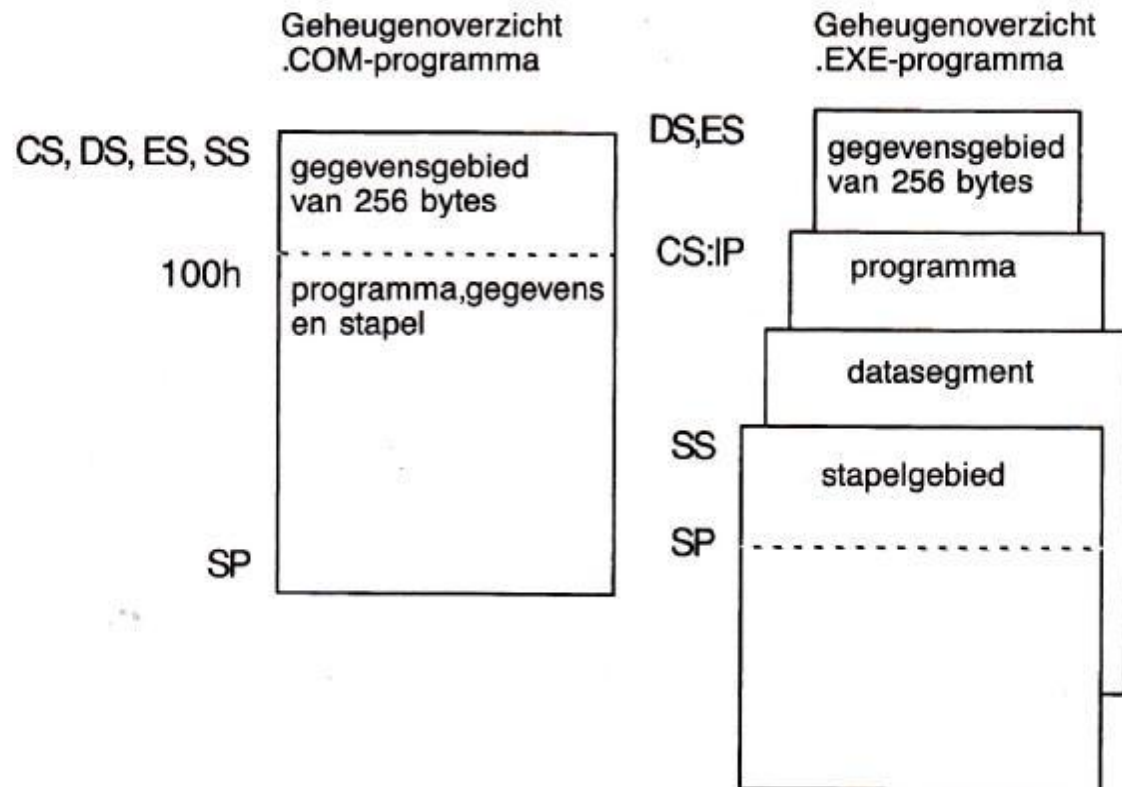
De code voor .COM-bestanden moet altijd beginnen op een offset 100h in het codesegment om ruimte over te houden voor dit 256 bytes lange kladgebied aan het begin. Dat is anders dan bij het .EXE-bestand, waarvan de code begon op IP = 0000, omdat het codesegment 100h bytes na het begin van het gebied in het geheugen begon. U zult misschien nog weten dat we in ons .COM-bestand in hoofdstuk 10 expliciet een pseudo-op ORG 100h aan het begin van ons programma moesten zetten om 100h bytes te reserveren. De pseudo-op ORG 100h zet de oorsprong (*ORiGin*) van onze code op 100h. Dat is het enige wat hij doet, maar we zullen de ORG 100h in onze

bestanden blijven gebruiken, omdat we in het verdere verloop van dit boek steeds .COM-bestanden zullen gebruiken.

We hebben u hier een .EXE-bestand laten zien om u iets over segmenten te kunnen leren. Verderop leert u er meer over, maar we zullen van nu af .COM-bestanden gebruiken, omdat ze kleiner zijn en sneller in het geheugen worden geladen. Hoe dat komt, ziet u in het laatste hoofdstuk, maar we gaan nu door met de pseudo-ops voor segmenten.

11.2 Pseudo-ops voor segmenten

We hebben hier verschillende pseudo-ops te bespreken : SEGMENT, ENDS, ASSUME en de NEAR- en FAR-aanroepen van de pseudo-op PROC. We moeten ook de instructies CALL en RET nader bekijken. Als we dat allemaal hebben gehad, gaan we meer over de INT-instructie leren en zien hoe die overeenkomt met een CALL-instructie. Maar we beginnen vooraan, met SEGMENT en ENDS.



Afb. 11-4. Vergelijking .COM- en .EXE-programma's.

De pseudo-ops SEGMENT en ENDS hebben veel weg van de pseudo-ops PROC en ENDP die we in hoofdstuk 9 tegenkwamen. We definiëren een segment door een deel

van de broncode te omgeven door het opdrachtenpaar `SEGMENT/ENDS`, net zoals we een procedure definieerden met het paar `PROC/END`. De naam vóór de pseudo-op `SEGMENT` is een label.

We zullen dit label gebruiken in hoofdstuk 13, wanneer we ons bronbestand opdelen in meerdere bronbestanden en twee segmenten — een datasegment en een codesegment. Met twee segmenten kunnen we gemakkelijk de variabelen in het geheugen scheiden van ons programma. Over variabelen in het geheugen leert u in hoofdstuk 13 ook meer, en we zullen ook nog delen aan de pseudo-op `SEGMENT` toevoegen. Er zijn echter talloze details bij dit onderwerp te bespreken, en we zullen daar niet veel tijd aan besteden. U kunt de informatie in de handleiding van uw assembler opzoeken als u wilt.

11.3 De pseudo-op **ASSUME**

De pseudo-op `ASSUME` is iets lastiger dan `SEGMENT`. Ze verschaft de assembler informatie over segmenten en over hoe we de segmentregisters willen gebruiken. Voor een goed begrip van `ASSUME` is inzicht nodig in de manier waarop de assembler labels en variabelenamen bijhoudt. Steeds als u een label creëert, bijvoorbeeld een procedure (zoals `SCHRIJF__TEK PROC NEAR`) of een geheugenvariabele, onthoudt de assembler *behalve* de naam nog andere informatie: het type (procedure, byte, woord, enz.), het adres van de naam en het segment waarin hij gedefinieerd is. Bij dit laatste stuk informatie is `ASSUME` betrokken.

De assembler neemt niet automatisch aan dat alle procedures van een programma in hetzelfde segment staan. In veel gevallen, zoals voor een groot programma als Lotus 1-2-3, is dat niet het geval. Zulke programma's gebruiken een aantal verschillende codesegmenten. Dus terwille van de algemene toepasbaarheid moeten we de assembler informatie verschaffen in de vorm van `ASSUME`-opdrachten die de assembler vertellen naar welke segmenten de segmentregisters wijzen.

Bekijk bijvoorbeeld eens de `ASSUME`-opdracht die we in vorige hoofdstukken hebben gebruikt:

```
ASSUME CS:CODE_SEG
```

Deze `ASSUME`-opdracht vertelt de assembler dat het `CS`-register wijst naar het codesegment dat we `CODE__SEG` hebben genoemd. Zonder deze informatie zou de assembler niet weten wat hij moet doen wanneer we proberen een label (zoals in `CALL SCHRIJF__TEK`) te gebruiken en met de melding *No or unreachable CS* zeggen dat hij geen idee heeft in welk segment we op dat moment zitten.

Omdat het `CS`-register altijd wijst naar de code die we uitvoeren, lijkt het misschien wat vreemd dat de assembler klaagt bij het ontbreken van een `ASSUME`-opdracht. Eigenlijk zouden we de `ASSUME`-opdracht ook niet nodig hebben als er niet de kwestie was van *segment-overrides* (segment-opheffingen)

De 8088 leest gegevens (zoals `MOV AL,EEN__VARIABELE`) normaliter uit het datasegment (`DS`) in. Maar hij kan ook informatie lezen uit elk ander segment, zoals het codesegment (`CS`), door gebruik te maken van een segment-override. En daarom heeft de assembler nu de pseudo-op `ASSUME` nodig: om te weten welk segmentregister hij moet gebruiken wanneer u uit het geheugen leest of ernaar schrijft.

Maakt u geen zorgen als u deze uitleg van ASSUME niet helemaal hebt begrepen. We zullen er tot aan hoofdstuk 29 zeer weinig gebruik van maken. Daarin leren we meer over zowel de pseudo-op ASSUME als over segment-overrides, wanneer we het gaan hebben over programma's met meer segmenten.

De overige informatie in dit hoofdstuk hoeft u alleen te lezen als u er belang in stelt, omdat we er in dit boek verder geen gebruik van zullen maken. U kunt de volgende twee paragrafen overslaan en later lezen als u het allemaal moeilijk vindt en graag weer aan het programmeren wilt.

11.4 NEAR- en FAR-aanroepen

Laten we even een stapje terugdoen en de CALL (aanroep)-instructies die we in de vorige hoofdstukken hebben gebruikt, wat nader bekijken, met name het programmaatje van hoofdstuk 7 waarin de CALL-instructie voor het eerst voorkwam. We schreven toen een heel kort programmaatje dat er als volgt uitzag (zonder de procedure op 200h)

3985:0100 B241	MOV	DL,41
3985:0102 B90A00	MOV	CX,000A
3985:0105 E8F800	CALL	0200
3985:0108 E2FB	LOOP	0105
3985:010A CD20	INT	20

U ziet aan de machinecode links dat de CALL-instructie slechts drie bytes beslaat (E8F800). De eerste byte (E8h) is de CALL-instructie, en de laatste twee bytes vormen een offset. De 8088 berekent het adres van de routine die we aanroepen door deze offset van 00F8h (de 8088 slaat de lage byte van een woord immers vóór de hoge byte in het geheugen op, dus we moeten de bytes omdraaien) op te tellen bij het adres van de volgende instructie (108h in ons programma). In dit geval krijgen we dan $F8h + 108H = 200H$. Precies wat we verwachtten.

Dat deze instructie een enkel woord voor de offset gebruikt, betekent dat CALLs zich slechts over één enkel segment, van 64K, kunnen uitstrekken. Maar hoe kunnen we dan een programma als Lotus 1-2-3 schrijven, dat groter dan 64K is? Door FAR-aanroepen in plaats van NEAR-aanroepen te gebruiken.

NEAR-aanroepen bestrijken, zoals we zagen, slechts één enkel segment. Met andere woorden, ze veranderen het IP-register zonder dat het CS-register erdoor wordt gewijzigd. En om die reden worden ze soms *intrasegmentale* aanroepen genoemd.

Maar we kunnen ook FAR-aanroepen hebben, die zowel het CS- als het IP-register beïnvloeden. Dergelijke CALLs worden vaak aangeduid als *intersegmentaal*, omdat ze procedures in andere segmenten aanroepen.

Met deze twee versies van de CALL-instructie gaan twee versies van de RET-instructie gepaard.

De NEAR-aanroep zet, zoals we in hoofdstuk 7 zagen, één enkel woord op de stapel voor het adres waarnaar hij moet terugkeren. En de overeenkomstige RET-instructie haalt dit woord weer van de stapel en zet het in het IP-register.

In het geval van FAR CALLs en FAR RETs is één woord niet voldoende omdat het om een ander segment gaat. Met andere woorden, we moeten een terugkeeradres van twee woorden op de stapel bewaren: een woord voor de instructiewijzer (IP) en het

andere voor het codesegment (CS). De FAR RET haalt dan de twee woorden van de stapel — één voor het CS-register en het andere voor IP.

Nu komen we bij een lastig punt. Hoe weet de assembler nu welk van deze twee CALLs en RETs hij moet gebruiken? Wanneer moet hij de FAR CALL en wanneer de NEAR CALL gebruiken? Hier nemen de pseudo-ops NEAR en FAR het bevel over.

Bekijk als voorbeeld eens het volgende programma:

```
PROC_EEN      PROC    FAR
    .
    .
    .
    RET
PROC_EEN      ENDP

PROC_TWEE     PROC    NEAR
    CALL      PROC_EEN
    .
    .
    .
    RET
PROC_TWEE     ENDP
```

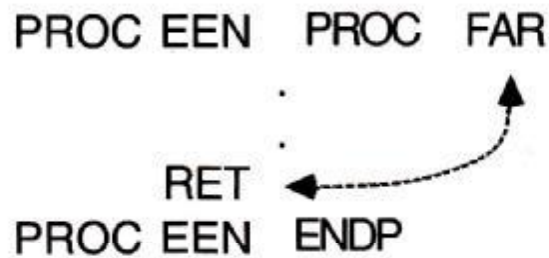
Wanneer de assembler de instructie CALL PROC_EEN ziet, zoekt hij in zijn tabel de definitie van PROC_EEN die, in dit geval, PROC_EEN PROC FAR is. De definitie maakt duidelijk of de procedure nabij of veraf is.

```
PROC TWEE     PROC    NEAR
    CALL      PROC EEN
    .
    .
    .
    RET
PROC TWEE     ENDP

PROC EEN      PROC    FAR
    .
    .
    .
    RET
PROC EEN      ENDP
```

Afb. 11-5. De assembler produceert een FAR CALL.

In het geval van een NEAR-procedure genereert de assembler een NEAR CALL. En omgekeerd genereert hij een FAR CALL als de procedure die u oproept is gedefinieerd als een FAR-procedure. Met andere woorden, de assembler gebruikt de defini-



Afb. 11-6. De assembler produceert een FAR RET.

tie van de procedure die u *aanroept* om vast te stellen welk type CALL-instructie er nodig is.

Voor de RET-instructie kijkt de assembler echter naar de definitie van de procedure die de RET-instructie bevat. In ons programma zal de RET-instructie voor PROC_EEN een FAR RET zijn omdat PROC_EEN is gedeclareerd als een FAR-procedure. Op dezelfde manier is de RET in PROC_TWEE een NEAR RET.

11.5 Meer over de INT-instructie

De INT-instructie heeft veel weg van een CALL-instructie, maar er is een klein verschil tussen. INT komt van het woord *interrupt* (onderbreking). Een interrupt is een extern signaal waardoor de 8088 een procedure uitvoert en dan terugkeert naar wat hij deed voor hij de interrupt binnenkreeg. Een INT-instructie onderbreekt de 8088 niet, maar wordt wel behandeld alsof dat het geval is.

Wanneer de 8088 een interrupt binnenkrijgt, moet hij meer informatie op de stapel zetten dan alleen de twee woorden voor het terugkeeradres. Hij moet de waarde van de statusvlaggen — de overdrachtvlag, de nulvlag, enz. — opslaan. Deze waarden worden bewaard in een woord dat bekend staat als het vlaggenreger, en de 8088 zet deze informatie vóór het terugkeeradres op de stapel. De reden dat we de statusvlaggen moeten bewaren, is de volgende:

Uw IBM-PC reageert regelmatig op een aantal verschillende interrupts. De 8088 binnenin uw IBM-PC ontvangt bijvoorbeeld per seconde 18,2 keer een interrupt van de klok. Elke van deze interrupts heeft als gevolg dat de 8088 stopt met wat hij aan het doen is en een procedure uitvoert om de klokimpulsen te tellen.

Stel u nu eens voor dat zo'n interrupt zich voordoet tussen deze twee programma-instructies in:

```

CMP      AH,2
JNE      NIET_2
  
```

Laten we aannemen dat AH = 2, zodat de nulvlag na de CMP-instructie gezet zal zijn, wat betekent dat de JNE-instructie niet naar NIET_2 springt.

Stel nu dat de klok de 8088 tussen deze twee instructies onderbreekt. Dat betekent dat de 8088 de interrupt-procedure uitvoert vóór hij de nulvlag controleert (met de JNE-instructie). Als de 8088 de vlaggenreger niet zou bewaren en herstellen, zou de JNE-instructie vlaggen gebruiken die door de interrupt-procedure zijn gezet, *niet* die van onze CMP-instructie. Om dergelijke rampen te voorkomen, bewaart en her-

stelt de 8088 *altijd* het vlaggenregister voor interrupts. Een interrupt bewaart de vlaggen, en een IRET (*Interrupt Return*)-instructie herstelt de vlaggen aan het eind van de interrupt-procedure.

Hetzelfde geldt voor een INT-instructie. Zo zal na uitvoering van de instructie:

INT 21

de stapel van de 8088 er als volgt uitzien:

Top van stapel →	oude IP (terugkeeradres deel 1)
	oude CS (terugkeeradres deel 2)
	oude vlaggenregister

(De stapel groeit in de richting van het lage geheugen, dus de top van de stapel staat onder het oude vlaggenregister.)

Als we een INT-instructie in een programma zetten, is de interrupt echter geen verrassing. Waarom willen we de vlaggen dan bewaren? Is het bewaren van vlaggen dan niet alleen zinnig wanneer we een externe interrupt hebben die zich op een onvoorspelbaar moment voordoet? Het antwoord blijkt nee te zijn. Er is een zeer goede reden om de vlaggen bij INT-instructies te bewaren en te herstellen. Sterker nog, Debug zou zonder dit niet mogelijk zijn.

Debug gebruikt een speciale vlag in het vlaggenregister, genaamd zekeringvlag (TF, *Trap Flag*). Deze vlag zet de 8088 in een speciale modus die bekend staat als single step-modus, die Debug gebruikt om programma's per instructie door te nemen. Wanneer de zekeringvlag gezet is, geeft de 8088 na elke instructie een INT 1 af.

De INT 1 maakt ook de zekeringvlag schoon, zodat de 8088 zich niet in single step-modus bevindt terwijl we binnen de INT 1-procedure van Debug zitten. Maar omdat INT 1 de vlaggen op de stapel heeft bewaard, leidt een IRET om terug te keren naar het programma dat we aan het debuggen waren tot herstel van de zekeringvlag. Daarna ontvangen we nog een INT 1-interrupt na de volgende instructie in ons programma. Dit is maar een voorbeeld van wanneer het zinnig is de vlaggenregisters te bewaren. Maar dit herstellen van de vlag is, zoals we straks zullen zien, niet altijd de juiste aanpak.

Sommige interrupt-procedures omzeilen het herstel van de vlaggenregisters. De INT 21-procedure in DOS, bijvoorbeeld, verandert de vlaggenregisters door het normale terugkeerproces kort te sluiten. Veel van de INT 21-procedures die schijfinformatie lezen of schrijven, keren terug met de overdrachtvlag gezet indien zich een of andere fout heeft voorgedaan (zoals geen schijf in de drive).

11.6 Interrupt-vectoren

Hoe komen deze interrupt-instructies nu aan de adressen voor de procedures? Elke interrupt-instructie heeft een interrupt-nummer, zoals de 21h in INT 21h. De 8088 vindt adressen voor interrupt-procedures in een tabel met *interrupt-vectoren*, die helemaal onderin het geheugen staat. Het twee woorden lange adres voor de INT 21h-procedure staat bijvoorbeeld op 0000:0084. We komen aan dit adres door het interrupt-nummer te vermenigvuldigen met 4 ($4 * 41h = 84h$), omdat we vier bytes, twee woorden, nodig hebben voor elke vector of procedure-adres.

Deze vectoren zijn van buitengewoon nut bij het toevoegen van mogelijkheden aan DOS, omdat ze ons in staat stellen aanroepen van interrupt-procedures te onderscheppen door de adressen in de vectortabel te wijzigen. We zullen dat in dit boek echter niet doen. Dergelijke trucs zijn voor ons nu nog te ingewikkeld.

Al deze ideeën en methoden zullen duidelijker worden wanneer we meer voorbeelden hebben gezien. Het boek zal van nu af aan merendeels vol staan met voorbeelden, dus u krijgt er voldoende te bestuderen. Als u zich wat beduusd voelt door alle nieuwe informatie, maakt u dan geen zorgen. In het volgende hoofdstuk doen we kalmer aan, heroriënteren ons en slaan het oude pad weer in.

11.7 Samenvatting

Zoals gezegd, bevatte dit hoofdstuk veel informatie. We zullen die niet allemaal gebruiken, maar we moesten wel leren hoe het zit met segmenten. In hoofdstuk 13 komen we toe aan modulair ontwerpen, en zullen we wat aspecten van segmenten in praktijk brengen om ons werk te vergemakkelijken.

We zijn dit hoofdstuk begonnen met te leren hoe de 8088 geheugen verdeelt in segmenten. Om nader inzicht in segmenten te krijgen, maakten we een .EXE-programma met twee verschillende segmenten. We zullen in dit boek geen .EXE-programma's gebruiken, maar het idee van segmenten werd met dit .EXE-programma wel duidelijk.

We zagen ook dat het kladgebied van 100h (256 bytes) aan het begin van onze programma's een kopie bevat van wat we op de commandoregel intikken. Ook deze kennis zullen we in dit boek verder niet gebruiken, maar het maakt wel beter duidelijk waarom DOS er zo'n groot stuk geheugen voor reserveert.

En ten slotte kwamen we bij de bespreking van de pseudo-ops SEGMENT, ENDS, ASSUME, NEAR en FAR. Dit zijn allemaal bewerkingen die ons helpen bij het werken met segmenten. We zullen in dit boek de kracht van deze pseudo-ops nauwelijks benutten, omdat onze .COM-programma's slechts één segment zullen gebruiken. Maar voor programmeurs die *gigantische* programma's in assembleertaal schrijven, zijn deze pseudo-ops van onschatbare waarde. Wie erin geïnteresseerd is, kan er alles over vinden in de handleiding van de macro-assembler.

Helemaal aan het eind van dit hoofdstuk leerden we meer over de achtergrond van onze nuttige INT-instructie. We kunnen nu wat rustiger aandoen en leren hoe we grotere en nuttigere programma's in assembleertaal kunnen *schrijven*.

12 Koerswijzigingen

12.1 Schijven, sectoren en Dskpatch 134

12.2 De aanpak 136

12.3 Samenvatting 137

We hebben onze neus in allerlei nieuwe en interessante zaken gestoken, en u hebt u hier en daar misschien afgevraagd of we maar wat doelloos rondliepen. Dat doen we natuurlijk niet. We kennen de omgeving nu goed genoeg om ons kompas te richten en een koers uit te zetten voor de rest van het boek. En dat gaan we in dit hoofdstuk doen: we gaan ons ontwerp van het Dskpatch-programma nader bekijken. Daarna doen we in het boek verder niets anders dan het ontwikkelen van Dskpatch, net zoals u later uw eigen programma's zult ontwikkelen.

We zullen de uiteindelijke versie van Dskpatch niet meteen laten zien; zo hebben we het niet geschreven. In plaats daarvan zullen we korte testprogramma's tonen, waarmee u het programma in elke fase waarin we het schrijven kunt testen. Daarvoor moeten we weten waar we heen willen. Vandaar de koerswijziging hier.

Omdat Dskpatch omgaat met informatie op onze schijven, beginnen we daarmee.

12.1 Schijven, sectoren en Dskpatch

De informatie op uw schijven is verdeeld in *sectoren*, die elk 512 bytes aan informatie bevatten. Een dubbelzijdige diskette die met DOS 2.0 of hoger is geformatteerd, heeft in totaal 720 sectoren, ofwel $720 * 512 = 368.640$ bytes. Als we die sectoren rechtstreeks konden bekijken, zouden we ook de directory rechtstreeks kunnen onderzoeken, of bestanden op de schijf bekijken. We kunnen dat niet zelf, maar wel met Dskpatch. Laten we Debug eens gebruiken om meer te leren over sectoren en een idee te krijgen van hoe we een sector met Dskpatch zullen afbeelden.

Debug heeft een opdracht genaamd L (*Load*, laad) om sectoren van de schijf in het geheugen te lezen, waar we de gegevens kunnen bekijken. Laten we bijvoorbeeld de directory eens bekijken die op sector 5 van een dubbelzijdige schijf staat. Laad sector 5 van de schijf in diskdrive A (dat is drive 0 voor Debug) op deze manier met de L-opdracht:

```
-L 100 0 5 1
```

Zoals u in afb. 12-1 ziet, laadt deze opdracht sectoren in het geheugen vanaf sector 5, tot aan een offset 100 van die sector binnen het datasegment. Om sector 5 te kunnen bekijken, gebruiken we een Dump-opdracht:

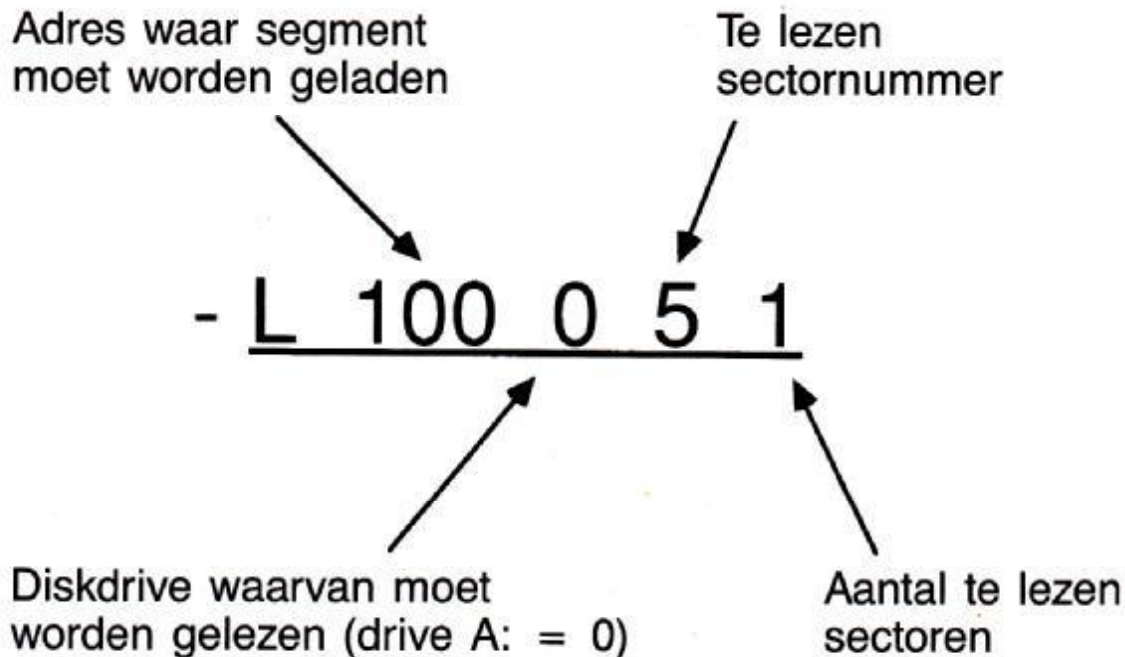
```
-D 100
```

396F:0100	4D 41 53 4D 20 20 20 20-45 58 45 20 00 00 00 00	MASM	EXE
396F:0110	00 00 00 00 00 00 00 20-50 0B 02 00 3E 4E 01 00	P...>N..
396F:0120	4C 49 4E 4B 20 20 20 20-45 58 45 20 00 00 00 00	LINK	EXE
396F:0130	00 00 00 00 00 00 00 20-50 0B 56 00 D4 AB 00 00	P.V.....
396F:0140	53 59 4D 44 45 42 20 20-45 58 45 20 00 00 00 00	SYMDEB	EXE
396F:0150	00 00 00 00 00 00 00 20-50 0B 81 00 9D 90 00 00	P.....
396F:0160	4D 41 50 53 59 4D 20 20-45 58 45 20 00 00 00 00	MAPSYM	EXE
396F:0170	00 00 00 00 00 00 00 20-50 0B A6 00 6A 46 00 00	P...jF..
-D			
396F:0180	43 52 45 46 20 20 20 20-45 58 45 20 00 00 00 00	CREF	EXE
396F:0190	00 00 00 00 00 00 00 20-50 0B B8 00 B4 3A 00 00	P.....
396F:01A0	4C 49 42 20 20 20 20 20-45 58 45 20 00 00 00 00	LIB	EXE
396F:01B0	00 00 00 00 00 00 00 20-50 0B C7 00 2C 70 00 00	P...,p..
396F:01C0	4D 41 4B 45 20 20 20 20-45 58 45 20 00 00 00 00	MAKE	EXE
396F:01D0	00 00 00 00 00 00 00 20-50 0B E4 00 EC 5E 00 00	P....^..


```

396F:01E0 45 58 45 50 41 43 4B 20-45 58 45 20 00 00 00 00  EXEPACK EXE ....
396F:01F0 00 00 00 00 00 00 00 20-50 0B FC 00 60 2A 00 00  .... P...!*..

```



Afb. 12-1. De Load-opdracht van Debug.

We zullen een dergelijke vorm ook gebruiken voor Dskpatch, maar met allerlei verbeteringen. Dskpatch wordt het equivalent van een volledige scherm-editor voor schijfsectoren. We zullen sectoren op het scherm kunnen afbeelden en de cursor over de sector-afbeelding verplaatsen en desgewenst getallen en tekens wijzigen. We zullen deze gewijzigde sector ook weer terug naar de schijf kunnen schrijven, wat ook de reden is dat we het de naam Disk Patch ('schijf-hersteller') hebben gegeven — of beter gezegd Dskpatch, omdat de naam niet langer mag zijn dan acht tekens.

Dskpatch is de drijfveer voor de procedures die we schrijven. Het is absoluut geen doel op zich. Door Dskpatch als voorbeeld in dit boek te gebruiken, kunnen we tevens allerlei procedures laten zien die u nuttig zult vinden bij het schrijven van uw eigen programma's. Dat betekent dat u veel algemene procedures zult zien voor uitvoer naar het scherm, bewerken van de afbeelding op het scherm, invoer via het toetsenbord enz.

Laten we een paar verbeteringen die we aan de sector-dump van Debug gaan aanbrengen eens nader bekijken. De afbeelding van Debug laat alleen de 'afdrukbare' tekens zien — 96 van de 256 verschillende tekens die een IBM-PC kan afbeelden. Hoe komt dat? Omdat MS-DOS, het broertje van PC-DOS, op allerlei verschillende computers draait. Sommige van die computers laten maar 96 tekens zien, en om die reden heeft Microsoft (dat Debug heeft geschreven) gekozen voor een versie van Debug die op alle machines werkt.

Dskpatch is bedoeld voor IBM Personal Computers en nauwe verwanten, dus we kunnen alle 256 verschillende tekens laten zien; dat vereist wel wat werk. Met de

DOS-functie 2 voor uitvoer van tekens kunnen we bijna alle tekens afbeelden, maar DOS geeft aan enkele daarvan een speciale betekenis, zoals aan 7, die een pieptoon geeft. Voor een code als 7 zijn er speciale tekens, en in deel 3 zullen ze zien hoe we die moeten afbeelden.

We zullen ook veel gebruik maken van de functietoetsen zodat we bijvoorbeeld de volgende sector kunnen zien door gewoon op F2 te drukken. En we zullen ook elke byte kunnen veranderen door de cursor naar die byte te verplaatsen en een nieuw getal te tikken. Het zal veel lijken op het gebruik van een tekstverwerker, waarmee we gemakkelijk tekens kunnen veranderen. Meer hierover zult u zien wanneer we Dskpatch geleidelijk aan opbouwen (Afb. 12-2 laat zien hoe het scherm er uit zal zien — een enorme verbetering vergeleken met Debug.)

Drive A	Sector 0	
	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	0123456789ABCDEF
00	34 90 49 42 4D 20 20 33 2E 33 00 02 02 01 00	IBM 3.3
10	02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00	Op 0 0
20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 12	
30	00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07	
40	BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00	
50	FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1	
60	06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10	
70	7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F	
80	7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03	
90	C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F	
A0	00 B8 01 02 E8 B3 00 72 19 8B FB B9 0B 00 BE D6	
B0	7D F3 A6 75 0D 8D 7F 20 BE E1 7D B9 0B 00 F3 A6	
C0	74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04	
D0	8F 44 02 CD 19 BE C0 7D EB EB A1 1C 05 33 D2 F7	
E0	36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00	
F0	07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38	

Druk op functietoets, of voer teken of hex-byte in:

Afb. 12-2. Voorbeeld van een Dskpatch-afbeelding.

12.2 De aanpak

In hoofdstuk 13 zullen we zien hoe we ons programma kunnen splitsen in veel verschillende bronbestanden. Daarna beginnen we in hoofdstuk 14 serieus aan Dskpatch te werken. Aan het eind zullen we negen bronbestanden voor Dskpatch hebben, die moeten worden gelinkt. En zelfs als u al die programma nu niet invoert en draait, dan hebt u toch altijd de bronbestanden als u er klaar voor bent, of als u algemene procedures wilt overnemen. U krijgt bij het doorlezen van de volgende hoofdstukken in elk geval een beter idee van hoe u lange programma's moet schrijven.

We hebben al verschillende nuttige procedures aangemaakt, zoals SCHRIJF_HEX, waarmee een byte als een hex-getal van twee cijfers wordt geschreven, en SCHRIJF_DECIMAAL, dat een getal in decimaal schrijft. Nu gaan we enkele pro-

gramma's schrijven om een geheugenblok op vrijwel dezelfde manier als met de D-opdracht van Debug af te beelden. We beginnen met het afbeelden van 16 bytes geheugen, een regel van de Debug-afbeelding, en werken dan toe naar een afbeelding van 16 regels van elk 16 bytes (een halve sector). Een hele sector past niet in een keer op het scherm bij de indeling die we hebben gekozen, zodat Dskpatch procedures bevat voor het verschuiven van een sector met behulp van ROM BIOS — geen DOS — interrupts. Dat komt echter veel later, nadat we een halve sector op een volledig scherm hebben afgebeeld.

Als we eenmaal 256 bytes geheugen kunnen dumpen, maken we een andere procedure om een sector van de schijf naar ons geheugengebied te kunnen lezen. We zullen een halve sector op het scherm dumpen, en dan met Debug ons programma kunnen veranderen om ook andere sectoren te kunnen dumpen. We hebben dan een functionele, maar niet erg aantrekkelijke, afbeelding, dus daarna gaan we haar verfraaien. Met nog wat meer werk en wat meer procedures, zullen we de halve sector-afbeelding zo ombouwen dat ze veel prettiger zal zijn om te zien. Het zal nog steeds geen volledige scherm-afbeelding zijn, dus ze zal net als de dump van Debug gewoon over het scherm rollen. Maar de volledige scherm-afbeelding komt daarna, en daarbij leren we ook de ROM BIOS-routines kennen, die ons in staat stellen de schermafbeelding te besturen, de cursor te verplaatsen en dat soort dingen. Daarna kunnen we nog meer ROM BIOS-routines leren gebruiken om alle 256 tekens af te drukken.

Vervolgens komen de toetsenbord-invoer en de procedures voor de opdrachten waarmee u met Dskpatch tot een dialoog kunt komen. Tegen die tijd zijn we ook weer aan een koerswijziging toe.

12.3 Samenvatting

We hebben nu genoeg van de toekomst gezien. U moet nu een wat beter idee hebben van waar we naartoe gaan, dus laten we nu doorgaan naar het volgende hoofdstuk, waarin we de basis van het modulair ontwerpen zullen leggen en leren hoe een programma wordt opgesplitst in verschillende bronbestanden. Daarna, in hoofdstuk 14, schrijven we enkele testprocedures om delen van het geheugen af te kunnen beelden.

13 Modulair ontwerpen

13.1 Afzonderlijk assembleren 140

13.2 De drie wetten van het modulair ontwerpen 143

13.3 Samenvatting 146

Zonder een modulaire opbouw zou Dskpatch niet erg leuk zijn om te schrijven. Zo'n opbouw maakt het ons veel gemakkelijker om iets langere programma's te schrijven. We zullen dit hoofdstuk gebruiken om wat elementaire regels van het modulair ontwerpen aan te geven, en ons de rest van het boek aan die regels houden. Laten we beginnen met te leren hoe een groot programma in veel verschillende bronbestanden kan worden verdeeld.

13.1 Afzonderlijk assembleren

In hoofdstuk 10 hebben we de procedure `SCHRIJF_DECIMAAL` toegevoegd aan `VIDEO_IO.ASM`, en er ook een korte testprocedure bijgevoegd met de naam `TEST_SCHRIJF_DECIMAAL`. We gaan die testprocedure nu eens uit `VIDEO_IO.ASM` halen en in een eigen bestand, `TEST.ASM`, zetten. Daarna assembleren we die twee bestanden afzonderlijk en linken ze tot één programma. Het bestand `TEST.ASM` ziet er zo uit.

Listing 13-1. Het bestand `TEST.ASM`

```
CODE_SEG      SEGMENT PUBLIC
               ASSUME  CS:CODE_SEG
               ORG     100h

               EXTRN   SCHRIJF_DECIMAAL:NEAR

TEST_SCHRIJF_DECIMAAL  PROC    NEAR
    MOV     DX,12345
    CALL    SCHRIJF_DECIMAAL
    INT     20h                ;terug naar DOS
TEST_SCHRIJF_DECIMAAL  ENDP

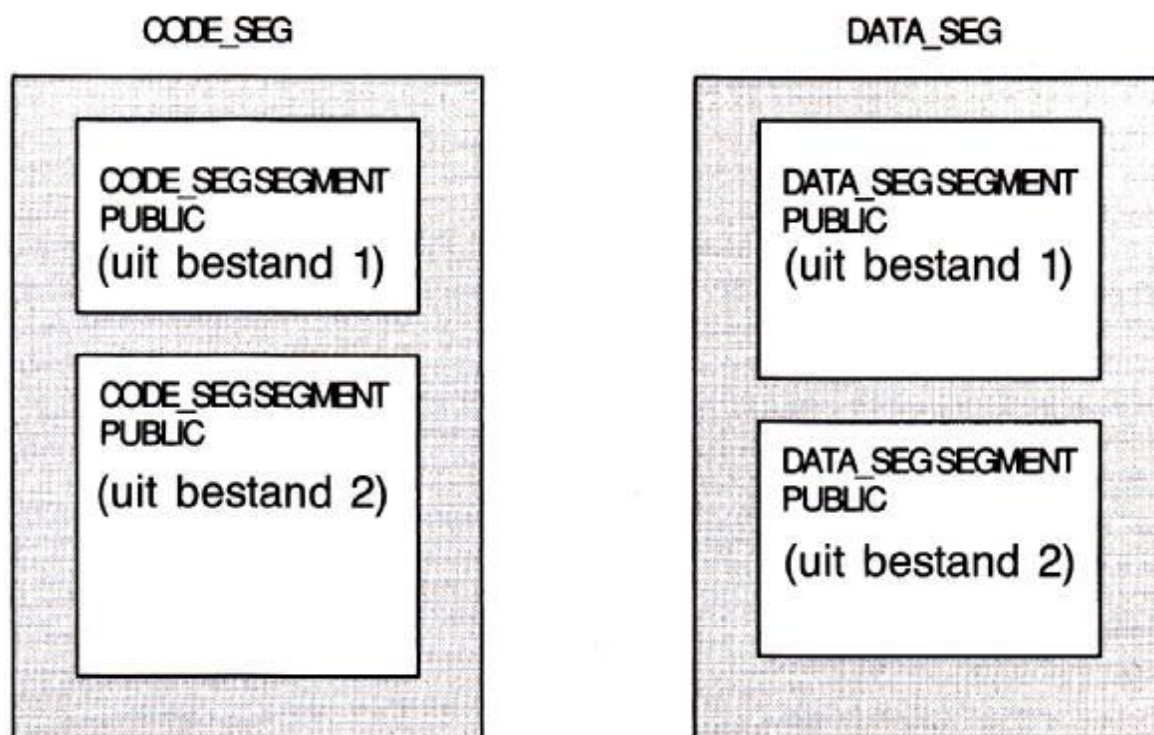
CODE_SEG      ENDS

               END      TEST_SCHRIJF_DECIMAAL
```

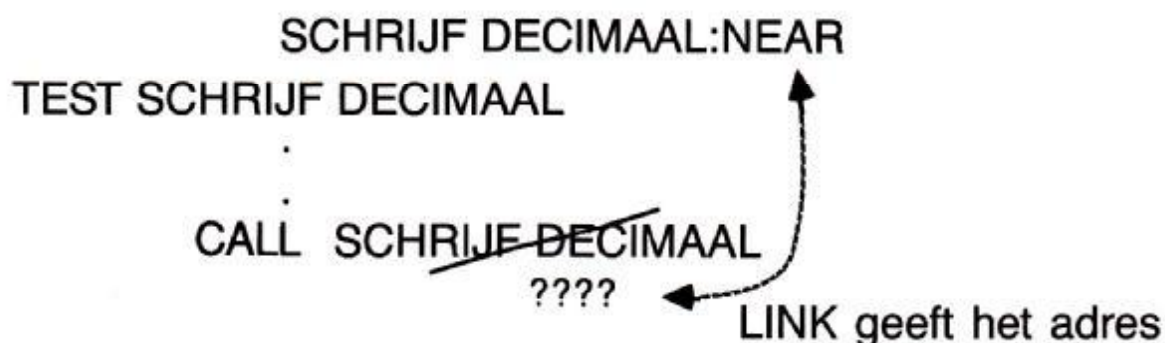
Het grootste deel van dit bronbestand hebben we al gezien, maar er staat iets nieuws in, dus laten we vooraan beginnen. Allereerst zien we nu het woord *PUBLIC* na `SEGMENT` staan. Hierdoor weet de assembler dat we dit segment (`CODE_SEG`) samen met alle andere segmenten van die naam willen samenvoegen in één segment — in dit geval het codesegment. De assembler geeft die informatie alleen maar door aan de linker die, zoals de naam al aangeeft, verschillende bestanden *linkt* (koppelt). De linker zet de verschillende delen van elk segment bij elkaar.

In ons bestand zien we nu de pseudo-op `EXTRN`. De opdracht `EXTRN SCHRIJF_DECIMAAL:NEAR` vertelt de assembler twee dingen: dat `SCHRIJF_DECIMAAL` in een ander, *extern*, bestand staat, en dat hij daarin als een NEAR-procedure is gedefinieerd, zodat hij in hetzelfde segment dient te staan. De assembler genereert voor deze procedure daarom een `NEAR CALL`; als we `FAR` na `SCHRIJF_DECIMAAL` hadden gezet, zou hij een `FAR CALL` genereren.

Dit zijn zo'n beetje de enige veranderingen die we nodig hebben voor afzonderlijke bronbestanden tot het moment waarop we gegevens in het geheugen beginnen op te slaan. Op dat punt zullen een ander segment voor gegevens ten tonele voeren (data-



Afb. 13-1. LINK zet segmenten uit verschillende bestanden bij elkaar.



Afb. 13-2. LINK kent de adressen voor externe namen toe.

segment). Laten we nu eerst `VIDEO_IO.ASM` wijzigen, en daarna deze twee bestanden assembleren en linken.

Haal de procedure `TEST_SCHRIJF_DECIMAAL` uit `VIDEO_IO.ASM`. We hebben die in `TEST.ASM` gezet, en hebben hem dus niet meer nodig in `Video_io`. Verwijder daarna de opdracht `ORG 100h` uit `Video_io`. Deze hebben we ook naar `TEST.ASM` verplaatst, die nu de eerste procedure in ons programma bevat. Zoals we in hoofdstuk 11 zagen, is de opdracht `ORG 100h` nodig om 256 bytes te bewaren voor het kladgebied aan het begin van ons programma — dat wil zeggen, voor `TEST_SCHRIJF_DECIMAAL` in het bronbestand `TEST.ASM`.

Vervolgens moeten we het woord *PUBLIC* achter *SEGMENT* zetten:

```
CODE_SEG      SEGMENT PUBLIC
```

zodat de linker weet dat hij dit segment met hetzelfde segment in TEST.ASM moet combineren.

Ten slotte moet *END TEST__SCHRIJF__DECIMAAL* aan het eind van VIDEO_IO.ASM worden veranderd in alleen *END*. Wederom omdat we de hoofdprocedure hebben overgezet naar TEST.ASM. De procedures in VIDEO_IO.ASM zijn nu *externe* procedures, meer niet. Dat wil zeggen, ze hebben als zodanig geen functie; ze moeten worden gelinkt met procedures die ze uit andere bestanden aanroepen. We hebben na de pseudo-op *END* in VIDEO_IO.ASM geen naam nodig omdat ons hoofdprogramma nu in TEST.ASM staat.

Als u klaar bent met deze veranderingen, moet uw bronbestand VIDEO_IO.ASM er ongeveer als volgt uitzien:

```
CODE_SEG      SEGMENT PUBLIC
               ASSUME  CS:CODE_SEG

               PUBLIC  SCHRIJF_HEX
               .
               .
               .
SCHRIJF_HEX    ENDP

               PUBLIC  SCHRIJF_HEX_CIJFER
               .
               .
               .
SCHRIJF_HEX_CIJFER    ENDP

               PUBLIC  SCHRIJF_TEK
               .
               .
               .
SCHRIJF_TEK      ENDP

               PUBLIC  SCHRIJF_DECIMAAL
               .
               .
               .
SCHRIJF_DECIMAAL    ENDP

CODE_SEG      ENDS

               END
```

met voorin een *ASSUME*.

Assembleer nu deze twee bestanden afzonderlijk net zoals u Video_io hebt geassembleerd. TEST.ASM weet nu alles wat hij over VIDEO_IO.ASM moet weten via de opdracht *EXTRN*. De rest komt wanneer we de twee bestanden linken.

U moet nu de bestanden TEST.OBJ en VIDEO_IO.OBJ hebben. Geef de volgende

opdracht om deze twee bestanden te koppelen tot een bestand met de naam TEST.EXE:

```
C>LINK TEST VIDEO_IO;
```

LINK voegt de procedures van deze twee bestanden samen tot een bestand dat het hele programma bevat. Hij gebruikt de eerste bestandsnaam die we als naam voor het ontstane .EXE-bestand hebben opgegeven, dus we hebben nu het bestand TEST.EXE.

Maak ten slotte een .COM-bestand aan door, net als eerder, *EXE2BIN TEST TEST.COM* te tikken. Dat is 't, we hebben nu van twee bestanden één programma gemaakt. Het uiteindelijke .COM-programma is identiek met de versie die we van het enkele bestand VIDEO_IO.ASM hadden gemaakt toen het de hoofdprocedure TEST_SCHRIJF_DECIMAAL bevatte.

We zullen van nu af veel gebruik maken van afzonderlijke bronbestanden en de waarde ervan zal duidelijker worden naarmate de procedures zich opstapelen. In het volgende hoofdstuk zullen we een testprogramma schrijven dat delen van het geheugen in hex dumpst. We zullen dan een eenvoudige testversie van een procedure schrijven vóór we de hele versie schrijven. Op die manier kunnen we zien hoe een goede uiteindelijke versie kan worden geschreven, en hoe we ons daarbij veel onrust en inspanning kunnen besparen.

Er zijn nog allerlei andere methoden om moeite te besparen. Wij noemen ze de *Drie wetten van het modulair ontwerpen*.

13.2 De drie wetten van het modulair ontwerpen

Het zijn niet echt *wetten*, het zijn suggesties. Maar we zullen ze het hele boek door toepassen. Bedenk uw eigen wetten als u wilt, maar houdt u hoe dan ook aan dezelfde. U hebt het veel gemakkelijker als u consequent bent. Ze luiden als volgt:

1. Bewaar en herstel alle registers, *tenzij* de procedure een waarde in dat register zet.
2. Wees consequent in welke registers u gebruikt om informatie door te geven. Bijvoorbeeld:
 - * DL, DX Geef byte- en woordwaarden door;
 - * AL, AX Retourneer byte- en woordwaarden;
 - * BX:AX Geef dubbele-woordwaarden door;
 - * DS:DX Heen- en terugkeeradressen;
 - * CX Herhalings- en andere telwaarden;
 - * CF Zetten bij foutconditie; in een van de registers, bijvoorbeeld AL of AX, moet een foutcode komen te staan.
3. Definieer *alle* externe interacties in het commentaar in de kop:
 - * bij ingang vereiste informatie;
 - * teruggestuurde informatie (gewijzigde registers);
 - * aangeroepen procedures;
 - * gebruikte variabelen (gelezen, geschreven, enz.).

Er is een duidelijke overeenkomst tussen modulair ontwerpen bij het programmeren en modulair ontwerpen in de techniek. Een electrotechnicus kan bijvoorbeeld een heel ingewikkeld apparaat maken uit dozen die verschillende dingen doen zonder te weten hoe elke doos werkt. Maar als elke doos een verschillende spanning en verschillende aansluitingen heeft, maakt dit gebrek aan uniformiteit het heel moeilijk voor de arme technicus, die op de een of andere manier moet zorgen dat elke doos een verschillende spanning en speciale aansluitingen krijgt. Geen leuke bezigheid, maar gelukkig zijn er normen die zorgen dat er maar een beperkt aantal standaardspanningen is. Dus hoeft hij bijvoorbeeld voor maar vier verschillende spanningen te zorgen, in plaats van een verschillende spanning voor elke doos.

Bij assembleerprogramma's zijn modulaire opbouw en standaardverbindingen even belangrijk, en daarom zullen we hier (zagezegd) de wetten voorschrijven en ons daar van nu af aan houden. Zoals u aan het einde van dit boek zult zien, zullen deze regels het ons veel gemakkelijker maken. Laten we ze nu eens in details bekijken.

Bewaar en herstel alle registers, *tenzij* de procedure een waarde in dat register zet.

Er zijn niet zó veel registers in de 8088. Door in het begin van de procedure registers te bewaren, kunnen we ze vrijelijk binnen die procedure gebruiken. Maar we moeten wel zorgen dat ze aan het einde van de procedure worden hersteld. U zult merken dat wij dat in al onze procedures doen, met voorin de procedures PUSH-instructies en POPs aan het eind.

De enige uitzondering bestaat uit procedures die bepaalde informatie aan de aanroepende procedure moeten teruggeven. Een procedure die bijvoorbeeld een teken van het toetsenbord leest, moet dat teken op de een of andere manier doorgeven. We zullen geen registers bewaren die we gebruiken om informatie door te geven.

Korte procedures helpen ook bij het probleem van het registertekort. Soms zullen we een procedure schrijven die maar één keer wordt gebruikt. Niet alleen helpt dit in verband met het tekort aan registers, het maakt het ook gemakkelijker om het programma te schrijven en vaak ook gemakkelijker te lezen. We zullen hier bij het schrijven van procedures voor Dskpatch meer van zien.

Wees consequent over welke registers u gebruikt om informatie door te geven.

Ons werk wordt eenvoudiger wanneer we bepaalde standaards hebben voor het uitwisselen van informatie tussen procedures. We zullen een register gebruiken voor het versturen van informatie, en een voor het ontvangen ervan. Voor lange stukken gegevens zullen we ook adressen moeten versturen, en hiervoor zullen we het registerpaar DS:DX gebruiken zodat onze gegevens overal in het geheugen kunnen staan. U leert hier meer over als we het over een nieuw segment voor gegevens hebben en het DS-register gaan gebruiken.

We reserveren het CX-register voor herhalingswaarden (teller). We zullen even verderop een procedure schrijven die een teken meermalen schrijft, zodat we tien spaties kunnen schrijven door deze procedure (SCHRIJF__TEK__N__KEER) aan te roepen met 10 in CX. We zullen het CX-register steeds gebruiken bij een herhalingsgetal of wanneer we een getal zoals het aantal ingelezen tekens van het toetsenbord willen doorgeven (dat doen we bij het schrijven van de procedure LEES__STRING).

Ten slotte zullen we bij elke fout de overdrachtvlag (CF, *carry flag*) gebruiken, en die leegmaken wanneer zich geen fout heeft voorgedaan. Niet alle procedures maken ge-

bruik van de overdrachtvlag. SCHRIJF_TEK werkt bijvoorbeeld altijd, dus er is geen enkele reden om een fout te willen doorgeven. Maar een procedure die naar de schijf schrijft, kan op talrijke fouten stuiten (geen schijf, schrijfbeveiliging, enz.). In dit geval zullen we een register gebruiken voor het doorgeven van een foutcode. Er is hier geen standaard voor, omdat DOS verschillende registers voor verschillende functies gebruikt. Hun fout, niet de onze.

Definieer *alle* externe interacties in het commentaar in de kop.

We hoeven niet te leren hoe een procedure werkt als we hem alleen maar willen gebruiken, en daarom hebben we voor elke procedure uitgebreid commentaar in de kop gezet. Deze kop bevat alle informatie die we nodig hebben. Er staat wat er in elk register moet staan vóór de procedure wordt aangeroepen, en welke informatie de procedure teruggeeft. De meeste procedures gebruiken registers voor hun variabelen, maar sommige procedures die we straks zullen zien, gebruiken variabelen in het geheugen. Het commentaar in de kop moet dan aangeven welke van deze geheugenvariabelen worden gelezen en welke veranderd. En ten slotte moet elk commentaar een opsomming van andere procedures bevatten die worden aangeroepen. Dit is een voorbeeld van een volledig commentaar in de kop waarin veel van deze informatie staat:

```

;-----;
; Dit is een voorbeeld van een volledig commentaar. Gewoonlijk bevat ;
; het een korte beschrijving van wat deze procedure doet. Deze ;
; procedure schrijft bijvoorbeeld de melding "Sector " op de eerste ;
; regel. ;
; ;
; DS:DX Adres van de melding "Sector " ;
; ;
; Roept aan: GANAAR_XY, SCHRIJF_STRING (aangeropen procedures) ;
; Leest: STATUSREGEL_NR (alleen gelezen variabelen) ;
; Schrijft: LOOS (gewijzigde geheugenvariabelen) ;
;-----;

```

Wanneer we een procedure willen gebruiken die we hebben geschreven, hoeven we alleen maar dit commentaar even te lezen om te zien hoe hij wordt gebruikt. U hoeft dan niet de hele procedure door te spitten om te zien wat hij doet.

Deze regels maken het programmeren in assembleertaal gemakkelijker, en we zullen ons er zeker aan houden, maar niet meteen de eerste keer — vaak ook helemaal niet. De eerste versie van een procedure of programma is een probeersel. Vaak weten we niet precies hoe we een programma moeten schrijven dat ons voor ogen staat, dus bij deze 'kladjes' zullen we het programma schrijven zonder ons te bekommeren om de wetten van het modulair ontwerpen. We ploegen er gewoon doorheen en krijgen dan iets wat werkt. Daarna kunnen we het spoor terugvolgen en de procedures keurig herschrijven zodat ze aan deze wetten voldoen.

Programmeren is een proces dat met horten en stoten gaat. Dit hele boek door zullen we laten zien met hoeveel haperingen Dskpatch is geschreven, maar we kunnen niet alles laten zien. Er is niet voldoende ruimte om alle versies te tonen die we hebben geschreven voor de uiteindelijke versie rond was. Onze eerste pogingen leken vaak erg weinig op de versie die u uiteindelijk ziet, dus maak u, als u programma's schrijft,

geen zorgen wanneer niet meteen alles werkt. Bereidt u erop voor dat u elke procedure moet herschrijven naarmate u beter in de gaten krijgt wat u eigenlijk wilt. In het volgende hoofdstuk zetten we een eenvoudig testprogramma op, dat een blok geheugen afdruckt. Het zal niet de uiteindelijke versie zijn; we zullen er nog meer proberen voor we tevreden zijn, en zelfs dan zullen we nog wijzigingen willen aanbrengen. De moraal van het verhaal is: een programma is nooit klaar, maar eens moet je ophouden.

13.3 Samenvatting

Dit was een hoofdstuk dat u voor later moet onthouden en dan gebruiken. We begonnen met te leren hoe een programma kan worden gesplitst in een aantal bronbestanden die we afzonderlijk kunnen assembleren en dan met de linker bij elkaar zetten. We gebruikten de pseudo-ops `PUBLIC` en `EXTRN` om de linker te vertellen dat er verbindingen tussen verschillende bronbestanden bestaan. `PUBLIC` geeft aan dat andere bronbestanden de procedures kunnen aanroepen die na `PUBLIC` staan, terwijl `EXTRN` de assembler vertelt dat de procedure die we willen gebruiken in een ander bestand staat.

We hebben `PUBLIC` ook na de `SEGMENT`-definitie gebruikt om de linker segmenten met dezelfde naam in verschillende bronbestanden bij elkaar te laten zetten. Daarna vervolgden we met de drie wetten van het modulair ontwerpen. Deze regels zijn bedoeld om u het programmeren gemakkelijker te maken, dus pas ze toe wanneer u zelf programma's schrijft, net zoals u ze in dit boek toegepast zult zien. U zult merken dat het schrijven, debuggen en lezen van programma's gemakkelijk gaat als u zich aan deze drie wetten houdt.

14 Geheugen dumpen

- 14.1 Adresseermodi 148**
- 14.2 Tekens aan de dump toevoegen 153**
- 14.3 256 bytes geheugen dumpen 155**
- 14.4 Samenvatting 159**

Vanaf nu zullen we ons concentreren op het opbouwen van Dskpatch op vrijwel dezelfde manier als we het oorspronkelijk hebben geschreven. Enkele instructies in de komende procedures zullen u niet bekend zijn; we zullen ze in het voorbijgaan kort uitleggen, maar voor meer informatie zult u een boek nodig hebben dat alle instructies in details bespreekt.

In plaats van alle 8088-instructies te behandelen, richten we ons op nieuwe terreinen, zoals de verschillende manieren waarop het geheugen kan worden geadresseerd, die we in het volgende hoofdstuk zullen bespreken. In deel 3 nemen we nog meer afstand van de details van instructies en krijgen we informatie te zien die specifiek is voor de IBM-PC en nauwe verwanten.

Laten we nu de *adresseermodi* bekijken door een kort testprogrammaatje te schrijven dat 16 bytes geheugen in hex-notatie dumpst. Om te beginnen moeten we leren hoe geheugen als variabele wordt gebruikt.

14.1 Adresseermodi

We hebben al twee adreseermodi gezien; ze staan bekend als *registeradressering* en *onmiddellijke adressering*. De eerste die we zagen, was de registermodus, die registers als variabelen gebruikt. Zo gebruikt de instructie:

```
MOV    AX,BX
```

de twee registers AX en BX als variabelen.

Daarna gingen we door naar de onmiddellijke (*immediate*) adreseermodus, waarin we een getal rechtstreeks in een register zetten, zoals in:

```
MOV    AX,2
```

Door deze instructie wordt de byte of het woord dat in het geheugen *onmiddellijk* na de instructie staat in een register gezet. In die zin is de MOV-instructie in dit voorbeeld één byte lang, met twee bytes extra voor de gegevens (0002):

```
396F:0100 B80200      MOV    AX,0002
```

De instructie is B8h, en de twee gegevensbytes (02h en 00h) komen daarna (onthoud dat de 8088 de lage byte, 02h, het eerst in het geheugen opslaat).

Nu gaan we zien hoe het geheugen als variabele kan worden gebruikt. In de onmiddellijke modus kunnen we de delen van het vaste geheugen onmiddellijk na die ene instructie lezen maar het geheugen niet wijzigen. Daarvoor hebben we andere adreseermodi nodig.

Laten we beginnen met een voorbeeld. Het volgende programma leest 16 geheugenbytes, één tegelijk, en laat elke byte in hex-notatie zien met een spatie tussen elk van de 16 hex-getallen. Voer het programma in, noem het TOON_SEC.ASM en assembleer het. Later zullen we nog wat kleine veranderingen in VIDEO_IO.ASM willen aanbrengen, maar we richten ons eerst op TOON_SEC.ASM:

Listing 14-1. Het nieuwe bestand TOON_SEC.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG      ;voeg twee segmenten samen
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
        ORG     100h

        EXTRN   SCHRIJF_HEX:NEAR
        EXTRN   SCHRIJF_TEK:NEAR

;-----;
; Dit is een eenvoudig testprogramma dat 16 bytes geheugen als ;
; hex-getallen dumpt, allemaal op een regel. ;
;-----;
TOON_REGEL    PROC      NEAR
        XOR     BX,BX                      ;maak BX 0
        MOV     CX,16                      ;dump 16 bytes
HEX_LUS:
        MOV     DL,SECTOR[BX]              ;haal 1 byte op
        CALL    SCHRIJF_HEX                ;dump deze byte in hex
        MOV     DL,' '                    ;zet een spatie tussen getallen
        CALL    SCHRIJF_TEK
        INC     BX
        LOOP    HEX_LUS
        INT     20h                        ;terug naar DOS
TOON_REGEL    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
        PUBLIC  SECTOR
SECTOR  DB     10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h ;testregel
        DB     18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
DATA_SEG      ENDS

        END     TOON_REGEL

```

Merk op dat we het datasegment (DATA__SEG) *na* het codesegment (CODE__SEG) hebben gezet. We hebben het aan het einde van het bestand gezet om te zorgen dat de linker de gegevens aan het einde van ons programma in het geheugen laadt. We hebben ook een paar nieuwe trucs aan dit programma toegevoegd, en daarom moeten we wat kleine veranderingen in VIDEO__IO.ASM aanbrengen. Verwijder eerst de ASSUME-opdracht uit Video__io en tik dan de volgende twee regels aan het begin van VIDEO__IO.ASM:

```

CGROEP  GROUP  CODE_SEG                      ;voeg twee segmenten samen
        ASSUME  CS:CGROEP

```

We zullen deze twee regels voortaan aan het begin van elk bestand zetten, met een lichte variant. Dan schrijven we:

```

CGROEP  GROUP  CODE_SEG, DATA_SEG          ;voeg twee segmenten samen
        ASSUME  CS:CGROEP

```


(met DATA__SEG erbij) steeds wanneer we zowel een codesegment als een datasegment in het bestand hebben.

De ASSUME vervangt hier de oude ASSUME, en we zullen verderop zien wat deze twee opdrachten in feite betekenen. Maar we gaan eerst even zien of ons nieuwe programma werkt. Assembleer Toon__sec en Video__io.

Nu kunnen we TOON__SEC.OBJ en VIDEO__IO.OBJ linken en het resultaat door Exe2bin halen. Gebruik dus eerst LINK om een .EXE-bestand genaamd TOON__SEC.EXE aan te maken. Het eerste bestandsnaam in de LINK-opdracht moet de naam zijn van het bestand dat de hoofdprocedure bevat (in dit geval Toon__sec), en achter de bestanden moet een puntkomma staan. Tik dus in:

```
C>LINK TOON__SEC VIDEO__IO;
```

Het linken gaat altijd op dezelfde manier, met meer namen voor de puntkomma wanneer we meer bestanden hebben, maar de hoofdprocedure moet altijd het eerste bestand zijn dat wordt genoemd.

Zet nu het .EXE-bestand in een .COM-bestand om door in te tikken:

```
C>EXE2BIN TOON__SEC TOON__SEC.COM
```

In het algemeen zien de twee stappen voor de bestanden *bestand1*, *bestand2*, enz., er zo uit:

```
LINK bestand1, bestand2, bestand3, ...;  
EXE2BIN bestand1, bestand1.COM
```

Draai nu het COM-bestand. Zorg dat u Exe2bin draait *voor* u *Toon__sec* draait. Anders draait u alleen maar de .EXE-versie van Toon__sec, en wie weet wat er dan gebeurt. In het ergste geval zult u dan uw computer uit moeten zetten, een minuutje wachten en dan weer aanzetten om de zaak te herstellen.

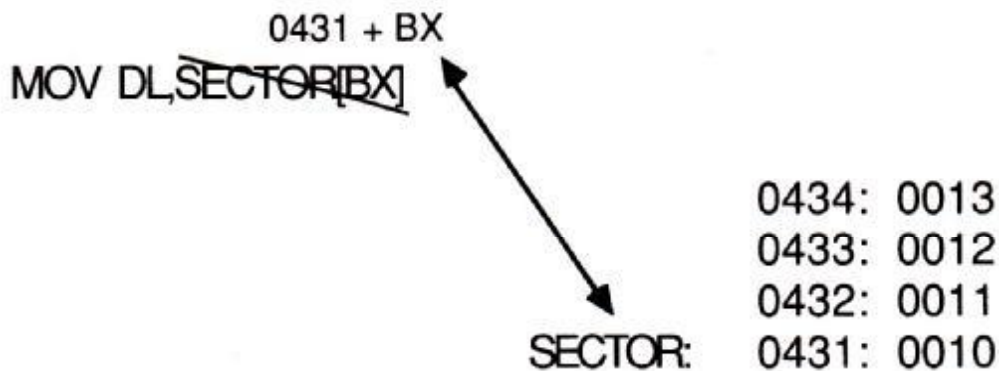
Als u dit niet ziet:

```
10 11 12 13 14 15 16 17 18 1A 1B 1C 1D 1E 1F
```

als u het programma draait, moet u terug en goed kijken of er ergens een fout staat. Laten we nu eens kijken hoe Toon__sec werkt. De instructie:

```
MOV     DL,SECTOR[BX]           ;haal 1 byte op
```

gebruikt een nieuwe adresseermodus die bekend staat als *indirecte geheugenadressering* — het adresseren van geheugen via het basisregister met een offset, ook wel, eenvoudiger, *base-relative* (indirect via basis) genoemd. Laten we eens kijken wat dit eigenlijk betekent:



Afb. 14-1. Vertaling van SECTOR[BX].

Als u Toon_sec bekijkt, zult u zien dat het label SECTOR in een segment met de naam DATA__SEG staat. Dit is een nieuw segment dat wordt gebruikt voor geheugenvariabelen. Iedere keer als we gegevens in het geheugen willen opslaan en lezen, zullen we wat ruimte voor dit segment vrijmaken. We komen zo terug op geheugenvariabelen, maar eerst moeten we iets over meer over segmenten vertellen.

De ASSUME DS:CGROEP vertelt de assembler waar hij geheugenvariabelen kan vinden. U had misschien al gedacht dat we ASSUME DS:DATA__SEG wilden hebben. Niet helemaal, want we willen een .COM-bestand aanmaken en dus maar één segment gebruiken. Toch is het gemakkelijk om er met twee te werken: een voor de code en een voor de gegevens. Op dit punt komt de pseudo-op GROUP ten tonele. GROUP groepeerde verschillende segmenten samen tot wat in feite één segment is, met de naam die we voor de pseudo-op GROUP opgeven. Dus de opdracht:

```
CGROEP GROUP CODE_SEG, DATA_SEG
```

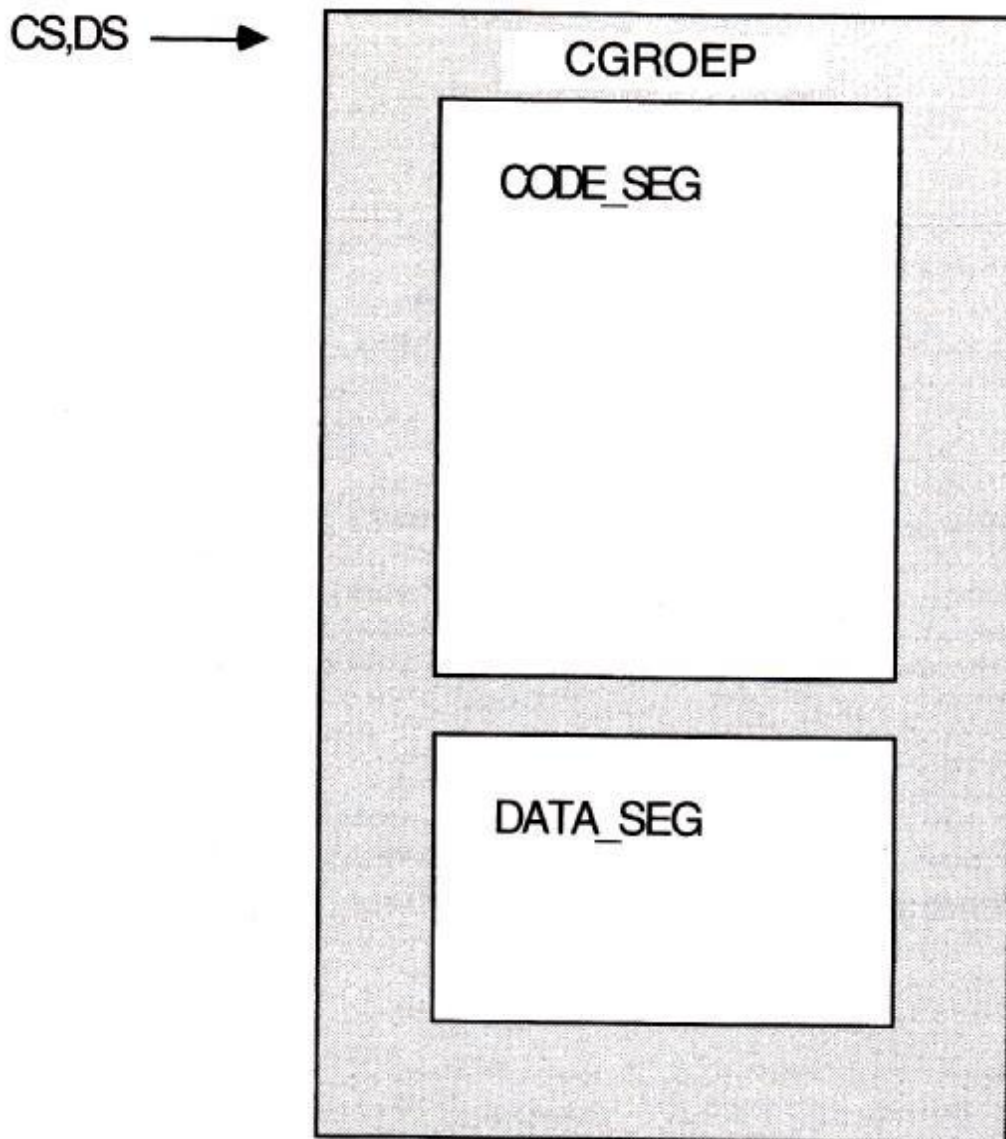
voegt de twee segmenten CODE__SEG en DATA__SEG samen tot één enkel segment van 64K met de naam CGROEP. De interne werking van groepen is weliswaar wat ingewikkelder, maar we hoeven verder geen details te weten. Als u die wel wilt weten, moet u de handleiding van uw macro-assembler naslaan. Maar wees gewaarschuwd: het is geen eenvoudige kost.

Het is nu tijd om terug te keren naar onze indirecte- adreseermodus. De twee regels:

```
SECTOR DB 10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h ;testregel
        DB 18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

maken 16 bytes geheugen in het datasegment vanaf SECTOR vrij, die de assembler omzet in een adres. DB betekent, zoals u zich misschien nog herinnert, *Define Byte*; de getallen na elke DB zijn beginwaarden. Dus als we TOON__SEC.COM voor de eerste keer starten, zal het geheugen vanaf SECTOR de waarden 10h, 11h, 12h, enz. bevatten. Als we geschreven hadden:

```
MOV DL, SECTOR
```

Afb. 14-2. GROUP behandelt meerdere sectoren als een enkel segment.

zou de instructie de eerste byte (10h) in het DL-register hebben gezet. Dit wordt *directe geheugenadressering* genoemd. Maar dat hebben we niet geschreven. In plaats daarvan hebben we [BX] na SECTOR gezet. Dat lijkt verdacht veel op een index van een array, zoals in de BASIC-opdracht:

$K = L(10)$

waardoor het 10de element van L in K komt te staan.

Onze MOV-instructie doet in feite hetzelfde. Het BX-register bevat een *offset* in het geheugen vanaf SECTOR. Dus als BX gelijk aan 0 is, zet MOVE DL,SECTOR[BX] de eerste byte (10h) in DL. Als BX 0A is, zet deze MOV-instructie het elfde byte (1Ah — we begonnen immers met 0) in DL.

Anderzijds zou de instructie MOV DX,SECTOR[BX] het zesde woord in DX hebben

gezet, omdat een offset van 10 bytes hetzelfde is als 5 woorden, en het eerste woord een offset nul is. (Voor enthousiastelingen: deze laatste MOV-instructie is niet toegestaan omdat SECTOR een byte-label is, terwijl DX een woord-register is. We zouden MOV DX,Word Ptr SECTOR[BX] moeten schrijven om de assembler te vertellen dat we SECTOR in deze instructie als een woord-label willen gebruiken.)

Er zijn nog vele andere adresseermodi; sommige ervan komen we verderop tegen, maar de meeste niet. Alle adresseermodi zijn samengevat in de volgende tabel:

Adresseermodus	Vorm adres	Gebruikt segmentregister
Register	register (b.v. AX)	Geen
Onmiddellijk	gegeven (b.v. 12345)	Geen
<i>Geheugenadresseermodi</i>		
Indirect register	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Indirect via basis*	label[BX]	DS
	label[BP]	SS
Direct geïndexeerd*	label[DI]	DS
	label[SI]	DS
Geïndexeerd via basis*	label[BX + SI]	DS
	label[BX + DI]	DS
	label[BP + SI]	SS
	label[BP + DI]	SS
String-opdrachten (MOVSW, LODSB, enz.)		Lezen uit DS:SI
		Schrijven naar ES:DI

* Label[...] kan worden vervangen door [verpl+...], waar *verpl* een verplaatsing is. We zouden dus [10+BX] kunnen schrijven en het adres zou dan 10+BX zijn.

14.2 Tekens aan de dump toevoegen

We zijn bijna klaar met de procedure voor een dump-afbeelding zoals die van Debug. Tot dusver hebben we de hex-getallen voor een regel gedumpt; de volgende stap zal zijn om de na de hex-getallen de tekens af te beelden. Het is niet zo ingewikkeld, dus hier is meteen de nieuwe versie van TOON_REGEL (in TOON_SEC.ASM), met een tweede lus voor het afbeelden van de tekens:

Listing 14-2. Veranderingen TOON_REGEL in TOON_SEC.ASM

```

TOON_REGEL      PROC      NEAR
                XOR      BX,BX                ;maak BX 0
                MOV      CX,16                ;dump 16 bytes
HEX_LUS:
                MOV      DL,SECTOR[BX]        ;haal 1 byte op
                CALL     SCHRIJF_HEX          ;dump deze byte in hex
                MOV      DL,' '               ;zet een spatie tussen getallen
                CALL     SCHRIJF_TEK
                INC      BX
                LOOP     HEX_LUS

                MOV      DL,' '               ;zet nog een spatie voor tekens
                CALL     SCHRIJF_TEK
                MOV      CX,16
                XOR      BX,BX                ;maak BX weer 0
ASCII_LUS:
                MOV      DL,SECTOR[BX]
                CALL     SCHRIJF_TEK
                INC      BX
                LOOP     ASCII_LUS

                INT      20h                  ;terug naar DOS
TOON_REGEL      ENDP

```

Assembleer dit, link het met Video__io, haal het door Exe2bin, en probeer het uit. Precies de afbeelding die we wilden. (Zie afb. 14-3.)

```

A:\> toon_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  >4!!78_1↑↓→L#AV
A:\>

```

Afb. 14-3. Uitvoer van TOON REGEL.

Probeer de gegevens eens zo te veranderen dat ook 0Dh tot en met 0Ah erbij staan. U zult dan een nogal vreemde afbeelding zien. Dat komt hierdoor: 0Ah en 0Dh zijn de tekens voor *line feed* (nieuwe regel) en *carriage return* (naar begin van regel). DOS ziet die tekens als opdrachten om de cursor te verplaatsen, maar wij zouden ze in deze afbeelding liever als gewone tekens zien. Daartoe moeten we SCHRIJF__TEK zo veranderen dat hij alle tekens afdruckt zonder er een speciale betekenis aan te hechten. We doen dat in deel 3; voorlopig veranderen we SCHRIJF__TEK een beetje, zodat hij een punt in plaats van de lage tekens (tussen 0 en 1Fh) afdruckt. Vervang de SCHRIJF__TEK in VIDEO__IO.ASM door deze nieuwe procedure:

```
A:\> toon_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
A:\>
```

Afb. 14-4. Gewijzigde versie van TOON REGEL.

Listing 14-3. Een nieuwe SCHRIJF_TEK in VIDEO_IO.ASM

```

PUBLIC  SCHRIJF_TEK
;-----;
; Deze procedure drukt een teken op het scherm af met behulp van de
; DOS-functie-aanroep. SCHRIJF_TEK vervangt de tekens 0 t/m 1Fh door
; een punt.
; DL      Op scherm af te drukken byte.
;-----;
SCHRIJF_TEK  PROC    NEAR
    PUSH    AX
    PUSH    DX
    CMP     DL,32                ;is vorige teken een spatie?
    JAE     IS_AFDrukBAAR       ;nee, dan zo afdrukken
    MOV     DL,'.'              ;ja, dan vervangen door een punt
IS_AFDrukBAAR:
    MOV     AH,2                ;uitvoer teken aanroepen
    INT     21h                 ;uitvoer teken in DL-register
    POP     DX                  ;herstel oude waarde in AX en DX
    POP     AX
    RET
SCHRIJF_TEK  ENDP

```

Probeer deze nieuwe procedure met Toon_sec en verander de gegevens in verschillende tekens om de grenscondities te controleren.

14.3 256 bytes geheugen dumpen

We zijn er nu in geslaagd om een regel, of 16 bytes, van het geheugen te dumpen. De volgende stap is om 256 bytes geheugen te dumpen. Dat is namelijk precies de helft van het aantal bytes in een sector, dus we gaan proberen een afbeelding te maken die een halve sector bevat. Er is nog veel te verbeteren; dit is nog maar een test-versie.

We hebben hier twee nieuwe procedures nodig, en een gewijzigde versie van TOON_REGEL. De nieuwe procedures zijn TOON_HALVE_SECTOR, die we al snel klaar zullen hebben en dan een halve sector laat zien, en STUUR_CRLF, die alleen de cursor naar het begin van de volgende regel stuurt (CRLF betekent *Carriage Return-Line Feed*, de twee tekens die de cursor aan het begin van de volgende regel zetten).

STUUR_CRLF is heel eenvoudig, dus laten we daarmee beginnen. Zet de volgende procedure in een bestand met de naam CURSOR.ASM.

Listing 14-4. Het nieuwe bestand CURSOR.ASM

```

CR          EQU      13          ;carriage return
LF          EQU      10          ;line feed

CGROEP     GROUP    CODE_SEG
           ASSUME    CS:CGROEP

CODE_SEG    SEGMENT PUBLIC

           PUBLIC   STUUR_CRLF
;-----;
; Deze routine stuurt alleen een carriage return en line feed naar het ;
; scherm met gebruik van de DOS-routines zodat het verschuiven van de ;
; cursor op de juiste wijze wordt verwerkt. ;
;-----;
STUUR_CRLF  PROC      NEAR
           PUSH      AX
           PUSH      DX
           MOV        AH,2
           MOV        DL,CR
           INT        21h
           MOV        DL,LF
           INT        21h
           POP        DX
           POP        AX
           RET
STUUR_CRLF  ENDP

CODE_SEG    ENDS

END

```

Deze procedure verstuurt een carriage return en line feed met behulp van de DOS-functie 2 voor het verzenden van tekens. De opdracht:

```
CR          EQU      13          ;carriage return
```

gebruikt de pseudo-op EQU om de naam CR gelijk (*EQU*al) aan 13 te maken. De instructie MOV DL,CR is hetzelfde als MOV DL,13. Zoals u in afb. 14-5 ziet, vervangt de assembler iedere CR die hij ziet door 13. Evenzo vervangt hij elke LF door 10.

```

CR EQU 13
.
.
.
MOV DL,CR 13

```

Afb. 14-5. Met de pseudo-op EQU kunnen we namen in plaats van getallen gebruiken.

Aan het bestand Toon_sec moet nu nogal wat worden veranderd. Dit is de nieuwe versie van TOON_SEC.ASM. Van nu af zullen we toevoegingen aan onze programma's tegen een grijze achtergrond laten zien; tekst die u dient te kennen, wordt afgebeeld in kleur:

Listing 14-5. De nieuwe versie van TOON_SEC.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG      ;voeg twee segmenten samen
        ASSUME  CS:CGROEP, DS:CGROEP
```

```
CODE_SEG  SEGMENT PUBLIC
        ORG     100h
```

```
        PUBLIC  TOON_HALVE_SECTOR
        EXTRN   STUUR_CRLF:NEAR

;-----
; Deze procedure beeldt een halve sector (256 bytes) af.
;
; Gebruikt:     TOON_REGEL, STUUR_CRLF
;-----
TOON_HALVE_SECTOR  PROC    NEAR
        XOR     DX,DX                ;begin bij begin van SECTOR
        MOV     CX,16                ;druk 16 regels af
HALVE_SECTOR:
        CALL    TOON_REGEL
        CALL    STUUR_CRLF
        ADD     DX,16
        LOOP    HALVE_SECTOR
        INT     20h
TOON_HALVE_SECTOR  ENDP

        PUBLIC  TOON_REGEL
        EXTRN   SCHRIJF_HEX:NEAR
        EXTRN   SCHRIJF_TEK:NEAR
```

```
;-----
; Deze procedure drukt een regel met gegevens, of 16 bytes, af; eerst
; in hex, daarna in ASCII.
;
; DS:DX  Offset in sector, in bytes.
;
; Gebruikt:  SCHRIJF_TEK, SCHRIJF_HEX
; Leest:     SECTOR
;-----
TOON_REGEL  PROC    NEAR
        XOR     BX,BX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                ;offset is handiger in BX
        MOV     CX,16                ;dump 16 bytes
        PUSH    BX                    ;bewaar de offset voor ASCII_LUS
HEX_LUS:
        MOV     DI,SECTOR[BX]        ;haal 1 byte op
        CALL    SCHRIJF_HEX          ;dump deze byte in hex
        MOV     DI,' '               ;zet een spatie tussen getallen
        CALL    SCHRIJF_TEK
```


Listing 14-5. *vervolg*

```

        INC     BX
        LOOP    HEX_LUS

        MOV     DL, ' '                ;zet nog een spatie voor tekens
        CALL    SCHRIJF_TEK
        MOV     CX, 16
        POP     BX                    ;haal offset in SECTOR terug
        XOR     BX, BX
ASCII_LUS:
        MOV     DL, SECTOR[BX]
        CALL    SCHRIJF_TEK
        INC     BX
        LOOP    ASCII_LUS
        POP     DX
        POP     CX
        POP     BX
        RET
        INT     20h
TOON_REGEL    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
                PUBLIC SECTOR
SECTOR        DB      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h ;testregel
                DB      18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
SECTOR        DB      16 DUP (10h)
                DB      16 DUP (11h)
                DB      16 DUP (12h)
                DB      16 DUP (13h)
                DB      16 DUP (14h)
                DB      16 DUP (15h)
                DB      16 DUP (16h)
                DB      16 DUP (17h)
                DB      16 DUP (18h)
                DB      16 DUP (19h)
                DB      16 DUP (1Ah)
                DB      16 DUP (1Bh)
                DB      16 DUP (1Ch)
                DB      16 DUP (1Dh)
                DB      16 DUP (1Eh)
                DB      16 DUP (1Fh)
DATA_SEG      ENDS

                END    TOON_HALVE_SECTOR

```

De wijzigingen zijn allemaal vrij simpel. In TOON_REGEL hebben we een PUSH BX en POP BX rond de HEX_LUS gezet omdat we de aanvankelijke offset in ASCII_LUS nogmaals willen gebruiken. We hebben ook PUSH- en POP-instructies toegevoegd om alle registers die we binnen TOON_REGEL gebruiken, te bewaren en te herstellen. TOON_REGEL is nu eigenlijk zo'n beetje klaar; de enige veranderingen die nog nodig zijn, zijn van esthetische aard en bedoeld om spaties en grafi-

sche tekens toe te voegen zodat we een aantrekkelijke afbeelding krijgen; maar die komen later.

Als u het bestand linkt, denk er dan aan dat we nu drie bestanden hebben: Toon__sec, Video__io en Cursor. Toon__sec moet vooraan in de lijst staan. Nadat u de .EXE-versie door Exe2bin hebt gehaald, ziet u een afbeelding als in 14-6.

```
C>toon_sec
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 .....
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 .....
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 .....
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 .....
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 .....
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 .....
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 .....
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 .....
18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 .....
19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 .....
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A .....
1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B .....
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C .....
1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D .....
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E .....
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F .....
```

C>

Afb. 14-6. Uitvoer van Toon sec.

We krijgen nog meer bestanden voor we klaar zijn, maar we gaan nu eerst door naar het volgende hoofdstuk, waarin we een sector rechtstreeks van de schijf in het geheugen zullen inlezen voor we een halve sector op het scherm dumpen.

14.4 Samenvatting

We weten meer over de verschillende geheugenmodi voor het adresseren van geheugen en registers in de 8088-microprocessor. We hebben geleerd over indirecte geheugenadressering, die we voor het eerst hebben gebruikt om 16 bytes geheugen te lezen. We hebben indirecte geheugenadressering ook gebruikt in verschillende programma's die we in dit hoofdstuk hebben geschreven, te beginnen met ons programma dat 16 hex-getallen op het scherm afdrukt. Deze 16 getallen kwamen uit een gebied in het geheugen met de naam SECTOR, dat we even later hebben uitgebreid om een geheugendump van 256 bytes — een halve sector — te kunnen afbeelden.

En ten slotte kregen we dumps te zien zoals die op het scherm worden afgebeeld, niet zoals ze worden afgedrukt op papier. In de volgende hoofdstukken zullen we een nuttiger gebruik van deze schermdumps maken.

15 Een schijfsector dumpen

- 15.1 Het leven gemakkelijker maken 162**
- 15.2 Indeling van het Make-bestand 162**
- 15.3 Toon__sec opknappen 163**
- 15.4 Een sector lezen 165**
- 15.5 Samenvatting 169**

Nu we een programma hebben dat 256 bytes geheugen dumpt, kunnen we er wat procedures aan toevoegen waarmee een sector van de schijf wordt gelezen en vanaf SECTOR in het geheugen wordt gezet. Daarna zetten onze dumpprocedures de eerste helft van deze schijfsector op het scherm.

15.1 Het leven gemakkelijker maken

Met de drie bronbestanden die we in het laatste hoofdstuk hadden, wordt het leven wat ingewikkeld. Hebben we alle drie bestanden veranderd waarmee we bezig waren, of maar twee? U hebt ze waarschijnlijk alle drie geassembleerd, in plaats van na te gaan of u ze nog hebt veranderd na de laatste keer dat ze zijn geassembleerd.

Maar al onze bronbestanden assembleren wanneer we er maar één hebben veranderd, duurt nogal lang en gaat langer duren naarmate Dskpatch groter wordt. Wat we eigenlijk willen, is alleen de bestanden assembleren die we hebben gewijzigd.

Gelukkig is daar een mogelijkheid toe, als u één van de laatste versies van de Macro Assembler van Microsoft (of hun C-compiler hebt). Die hebben een programma genaamd *Make* dat precies doet wat we nodig hebben. Om het te gebruiken, maken we een bestand aan (dat we Dskpatch noemen) dat Make vertelt hoe hij zijn werk moet doen, en tikken dan gewoon:

```
C>MAKE DSKPATCH
```

Make assembleert dan alleen de bestanden die u hebt veranderd.

Het bestand dat u aanmaakt (Dskpatch), vertelt Make welke bestanden afhankelijk van andere bestanden zijn. Iedere keer als u een bestand verandert, houdt DOS de tijd bij waarop het bestand is veranderd (u kunt dat in de DIR-afbeelding zien). Make kijkt gewoon naar zowel de .ASM- als de .OBJ-versies van een bestand. Als de .ASM-versie een recentere wijzigingstijd heeft dan de .OBJ-versie, weet Make dat hij dat bestand weer moet assembleren.

Dat is alles, maar we moeten u voor een ding waarschuwen: Make werkt alleen goed als u steeds ijverig de datum en tijd voor DOS invoert wanneer u de computer aanzet (wanneer dat al niet automatisch in orde komt). Zonder die informatie zal Make niet altijd weten wanneer u wijzigingen in een bestand hebt aangebracht.

15.2 Indeling van het Make-bestand

De indeling van ons bestand, Dskpatch, dat we voor Make zullen gebruiken, is vrij eenvoudig. Het is een tekstbestand dat u met een tekst-editor kunt aanmaken. Het ziet er als volgt uit:

Listing 15-1. Het Make-bestand DSKPATCH

```
toon_sec.obj:  toon_sec.asm
               masm toon_sec;

video_io.obj:  video_io.asm
               masm video_io;
```

Listing 15-1. *vervolg*

```
cursor.obj:      cursor.asm
               masm cursor;

toon_sec.com:    toon_sec.obj video_io.obj cursor.obj
               link toon_sec video_io cursor;
               exe2bin toon_sec toon_sec.com
```

Elke ingang heeft links een bestandsnaam (vóór de dubbelepunt) en rechts een of meer bestandsnamen. Als een van de bestanden rechts (zoals TOON__SEC.ASM op de eerste regel) recenter is dan het eerste bestand (TOON__SEC.OBJ) zal Make alle inspringende opdrachten op de volgende regel uitvoeren. (N.B. U moet die commandoregels inspringen met een tab, niet met spaties, althans in versies van Make vóór versie 4.05 die bij versie 5.00 van de Microsoft Macro Assembler hoort. Nu is het ook toegestaan om zelfs helemaal niet meer in te springen. Wij doen dat wel.) Als uw assembler het Make-programma bevat, voer dan deze regels in het bestand Dskpatch (zonder uitbreiding) in en verander iets kleins in TOON__SEC.ASM. Tik dan:

```
C>MAKE DSKPATCH
```

U ziet dan iets als dit:

```
Microsoft (R) Program Maintenance Utility  Version 4.05
Copyright (C) Microsoft Corp 1984-1987.  All rights reserved.

      masm toon_sec;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

51526 + 292490 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Make heeft de kleinst mogelijke hoeveelheid werk verricht die nodig is om onze programma's te herschrijven.

Als u geen recente versie van de Microsoft Macro Assembler met Make hebt, zult u merken dat het programma het geld waard is om een nieuwe versie aan te schaffen. En u krijgt dan ook een mooie nieuwe versie van Debug. Die heet Symdeb (*Symbolic Debugger*), die we later zullen bekijken. Nu gaan we verder met Dskpatch.

15.3 Toon__sec opknappen

Zoals Toon__sec er nu uit ziet, bevat hij een versie van TOON__HALVE__SECTOR, die we hebben gebruikt als testprocedure, en de hoofdprocedure. Nu gaan we TOON__HALVE__SECTOR in een gewone procedure veranderen zodat we hem kunnen aanroepen vanuit een procedure die we Disk__io zullen noemen. Onze test-

procedure zal in Disk_io komen te staan, samen met een testversie van de procedure voor het lezen van een schijfsector.

Laten we eerst Toon_sec zo wijzigen dat hij tot een procedurebestand wordt, net als we met Video_io hebben gedaan. Verander de END TOON_HALFVE-SECTOR in alleen END, omdat onze hoofdprocedure nu in Disk_io staat. Verwijder dan de opdracht ORG 100h uit CODE_SEG, ook weer omdat we deze naar een ander bestand hebben overgebracht.

Omdat we een sector in het geheugen willen lezen vanaf SECTOR, hoeven we geen testgegevens te verschaffen. We kunnen alle 16 DB-opdrachten na SECTOR vervangen door de ene regel:

```
SECTOR DB      8192 DUP (0)
```

die 8192 bytes voor het opslaan van de sector reserveert.

Maar herinnert u zich nog dat we zeiden dat sectoren 512 bytes lang zijn? Waarom hebben we dan zo'n groot opslaggebied nodig? Het blijkt dat op sommige harde schijven (zoals die van 300 megabyte) de sectoren erg groot zijn. Deze grote sectoren komen bepaald niet elke dag voor, maar we willen toch zeker weten dat we geen sector inlezen die te groot is voor het geheugen dat we voor SECTOR hebben gereserveerd. Daarom hebben we voor alle zekerheid 8192 bytes voor SECTOR gereserveerd. In het verdere verloop van dit boek nemen we, met uitzondering van SECTOR, die we zo bespreken, aan dat sectoren een lengte van slechts 512 bytes hebben.

Wat we nu nodig hebben is een nieuwe versie van TOON_HALFVE_SECTOR. De oude versie is slechts een testprocedure die we hebben gebruikt om TOON_REGEL te testen. In de nieuwe versie willen we een offset in de sector hebben zodat we 256 bytes kunnen afbeelden vanaf elk willekeurig punt in de sector. Dat betekent dan onder meer dat we de eerste helft, de laatste helft of de middelste 256 bytes kunnen dumpen. We zetten die offset weer in DX. Dit is de nieuwe — en definitieve — versie van TOON_HALFVE_SECTOR in TOON_SEC:

Listing 15-2. De definitieve versie van TOON_HALFVE_SECTOR
in TOON_SEC.ASM

```

PUBLIC TOON_HALFVE_SECTOR
EXTRN  STUUR_CRLF:NEAR

;-----;
; Deze procedure beeldt een halve sector (256 bytes) af. ;
; ;
; DS:DX Offset in sector, in bytes -- moet veelvoud van 16 zijn ;
; ;
; Gebruikt: TOON_REGEL, STUUR_CRLF ;
;-----;
TOON_HALFVE_SECTOR PROC NEAR
    XOR     DX,DX
    PUSH    CX
    PUSH    DX
    MOV     CX,16 ;beeld 16 regels af
HALVE_SECTOR:
    CALL    TOON_REGEL
    CALL    STUUR_CRLF
    ADD     DX,16

```

Listing 15-2. *vervolg*

```

        LOOP    HALVE_SECTOR
        POP     DX
        POP     CX
        RET
        INT     20h
TOON_HALVE_SECTOR      ENDP

```

We gaan nu verder met onze procedure voor het lezen van een sector.

15.4 Een sector lezen

In deze eerste versie van LEES_SECTOR negeren we expres fouten, zoals wanneer er geen schijf in de diskdrive zit. Het is geen juiste aanpak, maar het is toch niet de definitieve versie van LEES_SECTOR. We kunnen in dit boek niet ingaan op de behandeling van fouten, maar de versie van Dskpatch die op de bij dit boek te bestellen diskette verkrijgbaar is, bevat wel foutafhandelingsprocedures. Voorlopig willen we echter alleen maar een sector van de schijf lezen. De testversie van DISK_IO.ASM ziet er zo uit:

Listing 15-3. Het nieuwe bestand DISK_IO.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
        ORG    100h

        EXTRN  TOON_HALVE_SECTOR:NEAR

;-----;
; Deze procedure leest de eerste sector op schijf A en dumpt de eerste ;
; helft van deze sector. ;
;-----;
LEES_SECTOR  PROC    NEAR
        MOV     AL,0           ;diskdrive A (nummer 0)
        MOV     CX,1           ;lees maar 1 sector
        MOV     DX,0           ;lees sector 0
        LEA     BX,SECTOR      ;waar deze sector moet worden
                                ;opgeslagen
        INT     25h            ;lees de sector
        POPF                    ;verwijder door DOS op stapel
                                ;gezette vlaggen
        XOR     DX,DX          ;maak offset binnen SECTOR gelijk
                                ;aan 0
        CALL    TOON_HALVE_SECTOR ;dump eerste helft
        INT     20h            ;terug naar DOS
LEES_SECTOR  ENDP

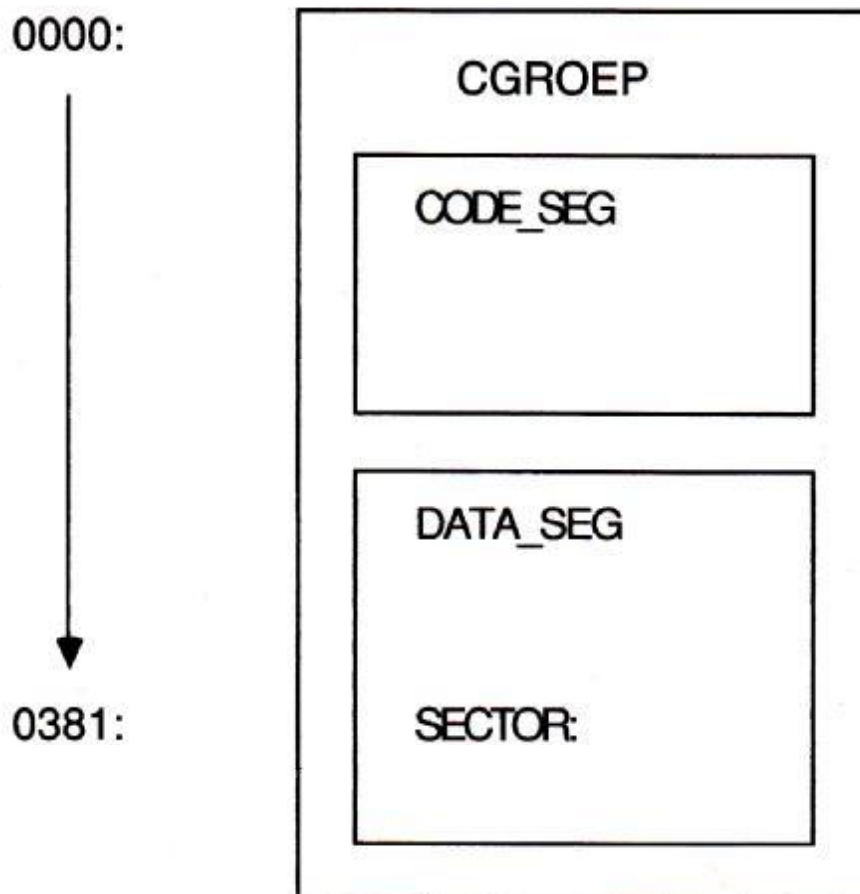
CODE_SEG      ENDS

```


Listing 15-3. *vervolg*

```
DATA_SEG      SEGMENT PUBLIC
               EXTRN      SECTOR:BYTE
DATA_SEG      ENDS

               END      LEES_SECTOR
```



~~LEA BX,SECTOR~~ ↔ MOV BX,0381

Afb. 15-1. LEA laadt het effectieve adres.

Deze procedure bevat drie nieuwe instructies. De eerste:

```
LEA    BX,SECTOR
```

zet het *adres*, ofwel de offset, van SECTOR (van het begin van CGROEP) in het BX-register; LEA betekent *Load Effective Address* (Laad Effectief Adres). Na deze LEA-instructie bevat DS:BX het volledige adres van SECTOR, en DOS gebruikt dit adres voor de tweede instructie, de aanroep van INT 25h, zoals we na nog enkele woorden over SECTOR zullen zien. (In feite laadt LEA de offset in het BX-register zonder het DS-register in te stellen; we moeten ervoor zorgen dat DS naar het juiste segment wijst.)

SECTOR staat niet in hetzelfde bronbestand als LEES_SECTOR. Hij staat verderop in TOON_SEC.ASM. Hoe vertellen we de assembler waar hij te vinden is? Met de pseudo-op EXTRN:

```
DATA_SEG      SEGMENT PUBLIC
               EXTRN      SECTOR:BYTE
DATA_SEG      ENDS
```

Deze instructies vertellen de assembler dat SECTOR is gedefinieerd in het DATA_SEG, dat in een ander bronbestand staat, en dat SECTOR een variabele met bytes is (in plaats van woorden). We zullen EXTRNs in de komende hoofdstukken nog veel gebruiken; ze vormen de manier waarop we dezelfde variabele in een aantal bronbestanden gebruiken. We moeten er alleen voor zorgen dat we onze variabelen op slechts één plaats definiëren.

```
DATA_SEG SEGMENT PUBLIC
               EXTRN      SECTOR:BYTE
DATA_SEG ENDS
```



Een byte-variabele.
LINK verschaft het adres.

Afb. 15-2. De pseudo-op EXTRN.

Terug naar onze instructie INT 25h. INT 25h is een speciale functie-aanroep van DOS voor het lezen van sectoren van een schijf. Wanneer DOS een aanroep van INT 25h ontvangt, gebruikt het de informatie in de registers als volgt:

AL	nummer van drive (0 = A, 1 = B, enz.);
CX	aantal in een keer te lezen sectoren;
DX	nummer van eerste te lezen sector (de eerste sector is 0);
DS:BX	overdracht-adres: waarheen de gelezen sectoren moeten worden geschreven.

Het nummer in het AL-register bepaalt de drive van waaruit DOS sectoren leest. Als AL = 0, leest DOS van drive A.

DOS kan met een enkele aanroep meer dan één sector lezen, en het leest het aantal sectoren dat in CX is opgegeven. We zetten hier CX op 1, zodat DOS maar één sector van 512 bytes leest.

We maken DX gelijk aan 0, zodat DOS de allereerste sector op de schijf leest. U kunt dit getal veranderen als u een andere sector wilt lezen; verderop doen we dat ook. DS:BX is het volledige adres van het gebied in het geheugen waar we willen dat DOS de sector(en) die het leest, opslaat. In dit geval hebben we DS:BX ingesteld op het adres van SECTOR, zodat we TOON__HALVE__SECTOR kunnen aanroepen voor het dumpen van de eerste helft van de eerste sector die van de schijf in drive A is gelezen.

Ten slotte zult u een POPF-instructie vlak na de INT 25h hebben gezien. We hebben het er al over gehad dat de 8088 een register bevat met de naam statusregister, dat de verschillende vlaggen bevat, zoals de nulvlag en de overdrachtvlag. POPF is een speciale POP-instructie die een woord in het statusregister zet. Waar is deze POPF-instructie voor nodig?

De instructie INT 25h zet eerst het statusregister en daarna het terugkeeradres op de stapel. Wanneer DOS van de INT 25h-interrupt terugkeert, laat het het statusregister op de stapel achter. DOS doet dat om bij terugkeer de overdrachtvlag te kunnen zetten wanneer zich een schijffout voordoet, zoals proberen te lezen van drive A: zonder dat er een schijf in zit. We zullen in dit boek niet op fouten controleren, maar we moeten wel zelf het statusregister van de stapel halen — vandaar de POPF-instructie. (N.B. INT 25h en INT 26h, die een sector naar een schijf *schrijft*, zijn de enige DOS-routines die het statusregister op de stapel achterlaten.)

Nu kunt u DISK__IO.ASM assembleren, en dan TOON__SEC.ASM opnieuw assembleren. Link daarna de vier bestanden Disk__io, Toon__sec, Video__io en Cursor, met eerst Disk__io. Of voeg, als u Make hebt, deze twee regels toe aan uw Dskpatch bestand:

```
disk_io.obj:    disk_io.asm
               masm disk_io;
```

en verander de laatste drie regels in:

```
disk_io.com:    disk_io.obj toon_sec.obj video_io.obj cursor.obj
               link disk_io toon_sec video_io cursor;
               exe2bin disk_io disk_io.com
```

Nadat u uw .COM-versie van Disk__io hebt aangemaakt, moet u een scherm zien dat er ongeveer uit ziet als in afb. 15-3:

```

A:\> disk io
EB 34 90 49 42 4D 20 20 33 2E 32 00 02 02 01 00 64EIBM 3.2....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 .p. ., 2 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0F ..... 3 12 11 1..
00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07 ..... 3 12 11 1..
BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00 7X,6+7.U.S+!q..
FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1 0/80=.t.&e.-e-f+
06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10 ..eG. .+!f=.rga.
7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F iy&. . . . .u?
7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03 iu7!q .z&.i. .i.
C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 96 HZ<.7!q. .i?i8u
00 B8 01 02 E8 AA 00 72 19 8B FB B9 0B 00 BE CD .j. .2-.r. .i?j. .j=
7D F3 A6 75 0D 8D 7F 20 BE D8 7D B9 0B 00 F3 A6 }-au.ia +}j. .<a
74 18 BE 6E 7D E8 61 00 32 E4 CD 16 5E 1F 8F 04 t.jn>8a.2Σ=.^,A.
8F 44 02 CD 19 BE B7 7D EB EB A1 1C 05 33 D2 F7 aD.=.j>8di. .3πz
36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00 6. .!L6<i7!u=iq.
07 A1 37 7C E8 40 00 A1 18 7C 2A 06 3B 7C 40 50 .i7!8e.i. !*.;!0P
A:\>

```

Afb. 15-3. Schermdump van DISK IO.COM

15.5 Samenvatting

Nu we onze vier verschillende bronbestanden hebben, wordt Dskpatch wat ingewikkelder. In dit hoofdstuk hebben we het programma Make bekeken, dat ons het leven vergemakkelijkt door alleen de bestanden te assembleren die we hebben veranderd. We hebben ook een nieuwe procedure geschreven: DISK__IO. Hij staat in een ander bronbestand dan SECTOR, zodat we een EXTRN in DISK__IO.ASM hebben gezet om de assembler van SECTOR op de hoogte te brengen en te vertellen dat SECTOR een byte-variabele is.

Ook hebben we de instructie LEA (*Load Effective Address*) leren kennen; we hebben haar gebruikt om het adres van SECTOR in het BX-register te laden.

DISK__IO gebruikt een nieuw INT-nummer, INT 25h, om sectoren van een schijf naar het geheugen te lezen. We hebben INT 25h gebruikt om een sector te lezen naar onze geheugenvariabele, SECTOR, om die met TOON__HALVE__SECTOR op het scherm te kunnen dumpen.

Verder hebben we geleerd over de POPF-instructie, die een woord van de stapel haalt en in het statusregister zet. We hebben deze instructie gebruikt om de vlaggen te verwijderen die DOS niet van de stapel haalde toen het terugkeerde van INT 25h.

Onze afbeelding van de halve sector ziet er nog niet aantrekkelijk uit, maar in het volgende hoofdstuk zullen we enkele van de grafische tekens van de IBM-PC gebruiken om het aanzien wat aantrekkelijker te maken.

16 Verfraaien van de sector-afbeelding

- 16.1 Grafische tekens toevoegen 172**
- 16.2 Adressen aan de afbeelding toevoegen 174**
- 16.3 Horizontale lijnen toevoegen 177**
- 16.4 Getallen aan de afbeelding toevoegen 181**
- 16.5 Samenvatting 183**

We zijn bij het laatste hoofdstuk van deel 2 aangekomen. Alles wat we tot dusver hebben gedaan, is van toepassing op MS-DOS en de 8088 (of de 8086 en andere familieleden van de 8088). In deel 2 beginnen we procedures te schrijven die specifiek voor de IBM Personal Computer en zijn naaste verwanten zijn.

Maar voor we verder gaan, gebruiken we dit hoofdstuk om nog allerlei procedures toe te voegen aan Video__io. We zullen ook TOON__REGEL in Toon__sec wijzigen. Al onze wijzigingen en toevoegingen hebben te maken met het uiterlijk van de weergave op het scherm. De meeste ervan zullen de afbeelding verfraaien, maar één ervan voegt nieuwe informatie toe: hij zet nummers aan de linkerkant die de adressen in de dump van Debug voorstellen. We beginnen met de grafische tekens.

16.1 Grafische tekens toevoegen

De IBM Personal Computer kent een aantal tekens voor het tekenen van lijnen die we kunnen gebruiken om kaders te maken rondom verschillende delen van onze dump-afbeelding. We zullen een kader om de hex-dump maken, en een rond de ASCII-dump. Deze verandering vereist weinig denkwerk, alleen wat inspanning. Voeg de volgende definities aan het begin van het bestand TOON__SEC.ASM in, tussen de pseudo-op ASSUME en de eerste pseudo-op SEGMENT, en laat een of twee regels voor en na deze definities leeg:

Listing 16-1. Toevoegen aan begin TOON__SEC.ASM

```
;-----;
; Grafische tekens voor randen van sector.      ;
;-----;
VERTICALE_BALK      EQU      0BAh
HORIZONTALE_BALK    EQU      OCDh
BOVEN_LINKS         EQU      0C9h
BOVEN_RECHTS        EQU      0BBh
ONDER_LINKS          EQU      0C8h
ONDER_RECHTS         EQU      0BCh
TOP_T_BALK           EQU      0CBh
BENEDEN_T_BALK       EQU      0CAh
TOP_STREEPJE         EQU      0D1h
BENEDEN_STREEPJE     EQU      0CFh
```

Dit zijn de definities voor de grafische tekens. Merk op dat we voor elk hex-getal een 0 hebben gezet zodat de assembler weet dat het om getallen en niet om labels gaat.

We hadden net zo eenvoudig hex-getallen als deze definities in onze procedure kunnen zetten maar de definities maken de procedure gemakkelijker te begrijpen. Vergelijk bijvoorbeeld de volgende instructies eens:

```
MOV     DL,VERTICALE_BALK
MOV     DL,0BAh
```

De meeste mensen vinden de eerste instructie duidelijker.

Dan is hier de nieuwe procedure TOON_REGEL om de verschillende delen van het scherm te scheiden met het VERTICALE_BALK-teken, nummer 186 (0BAh). U ziet de toevoegingen weer tegen een gekleurde achtergrond:

Listing 16-2. Veranderingen van TOON_REGEL in TOON_SEC.ASM

```
TOON_REGEL      PROC      NEAR
                PUSH      BX
                PUSH      CX
                PUSH      DX
                MOV        BX,DX                      ;offset is nuttiger in BX
                MOV        DL,' '                      ;schrijf verticale balk
                CALL        SCHRIJF_TEK
                MOV        DL,VERTICALE_BALK          ;teken linkerkant kader
                CALL        SCHRIJF_TEK
                MOV        DL,' '
                CALL        SCHRIJF_TEK
                MOV        CX,16                      ;schrijf nu 16 bytes
                PUSH      BX                          ;dump 16 bytes
                HEX_LUS:
                MOV        DL,SECTOR[BX]              ;haal 1 byte op
                CALL        SCHRIJF_HEX               ;dump deze byte in hex
                MOV        DL,' '                      ;zet een spatie tussen getallen
                CALL        SCHRIJF_TEK
                INC        BX
                LOOP        HEX_LUS
                MOV        DL,VERTICALE_BALK          ;schrijf verticale balk
                CALL        SCHRIJF_TEK
                MOV        DL,' '
                CALL        SCHRIJF_TEK
                MOV        CX,16
                POP        BX                          ;haal offset in SECTOR terug
                ASCII_LUS:
                MOV        DL,SECTOR[BX]
                CALL        SCHRIJF_TEK
                INC        BX
                LOOP        ASCII_LUS
                MOV        DL,' '                      ;teken rechterkant kader
                CALL        SCHRIJF_TEK
                MOV        DL,VERTICALE_BALK
                CALL        SCHRIJF_TEK
                POP        DX
                POP        CX
                POP        BX
                RET
TOON_REGEL      ENDP
```

Assembleer deze nieuwe versie van Toon_sec en link uw bestanden (denk eraan Disk_io vooraan in de reeks bestanden na de LINK opdracht te zetten). U ziet dan twee mooie verticale lijnen die de afbeelding in twee stukken verdelen, zoals u in afb. 16-1 kunt zien.


```

A:\> disk_io
EB 34 90 49 42 4D 20 20 33 2E 32 00 02 02 01 00 64E1BM 3.2....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 .p.u.2.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....3.8.11.1.
BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00 7x.6+7.U.S.+11.
FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1 1188= .t.&e.-e-r+
06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10 ..eG.1.+i|=,rga.
7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F iy&. . . . .u?
7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03 iu717 .z&.11.1.
C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 96 HZ< .7!7. .i?18u
00 B8 01 02 E8 AA 00 72 19 8B FB B9 0B 00 BE CD .7. .87.r.1y1. .3=
7D F3 A6 75 0D 8D 7F 20 BE D8 7D B9 0B 00 F3 A6 }<au.iΔ +#}1. .<a
74 18 BE 6E 7D E8 61 00 32 E4 CD 16 5E 1F 8F 04 t.7n}8a.2Σ=.^,A.
8F 44 02 CD 19 BE B7 7D EB EB A1 1C 05 33 D2 F7 AD.=.3p}88i. .3π~
36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00 6.11L6<1171u=17.
07 A1 37 7C E8 40 00 A1 18 7C 2A 06 3B 7C 40 50 .17180.i.1*.;10P

```

A:\>

Afb. 16-1. Schermdump met verticale lijnen.

16.2 Adressen aan de afbeelding toevoegen

Nu gaan we iets moeilijkers proberen: aan de linkerkant van de afbeelding de hex-adressen erbij zetten. Deze getallen worden de offset van het begin van de sector, dus het eerste getal is 00, het volgende 10, dan 20 enz.

Het is vrij eenvoudig, omdat we al beschikken over de procedure SCHRIJF_HEX die een getal in hex schrijft. Maar er is een probleem omdat de sector 512 bytes lang is: SCHRIJF_HEX drukt alleen hex-getallen van twee cijfers af, terwijl we drie hex-cijfers nodig hebben voor getallen die groter zijn dan 255.

De oplossing is als volgt. Omdat onze getallen tussen nul en 511 zullen liggen (0h en 1FFh), zal het eerste cijfer óf een spatie zijn, als het getal (bijvoorbeeld BCh) kleiner dan 100h is, of een 1. Dus als het getal groter is dan 255, drukken we gewoon een 1 af, gevolgd door het hex-getal van de lage byte. Anders drukken we eerst een spatie af. Dit zijn de toevoegingen aan TOON_REGEL waardoor dit driecijferige hex-getal aan de linkerkant wordt afgedrukt:

Listing 16-3. Toevoegingen TOON_REGEL in TOON_SEC.ASM

TOON_REGEL	PROC	NEAR	
PUSH	BX		
PUSH	CX		
PUSH	DX		
MOV	BX,DX		;offset is nuttiger in BX
MOV	DL,' '		
			;schrijf offset in hex
CMP	BX,100h		;is het eerste cijfer een 1?
JB	SCHRIJF_EEN		;nee, schrijf spatie die al DL staat
MOV	DL,'1'		;ja, zet dan '1' in DL voor uitvoer

Listing 16-3. *vervolg*

```

SCHRIJF_EEN:
    CALL    SCHRIJF_TEK
    MOV     DL,BL                      ;kopieer lage byte naar DL voor
                                         ;hex-uitvoer
    CALL    SCHRIJF_HEX
                                         ;schrijf verticale balk
    MOV     DL,' '
    CALL    SCHRIJF_TEK
    MOV     DL,VERTICALE_BALK          ;teken linkerkant kader

```

Het resultaat ziet u in afb. 16-2:

```

A:\> disk io
00 EB 34 90 49 42 4D 20 20 33 2E 32 00 02 02 01 00 64 IBM 3.2....
10 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 .p.u.2.....
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....3 4 5 6...
30 00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07 x.6+7.U.S.+i..
40 BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00 0/89=.t.&e.rê-r+
50 FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1 0/89=.t.&e.rê-r+
60 06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10 ..eG. ||.+i|=,rgâ,
70 7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F iyz&.....iü?
80 7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03 iü7!q ,z&.i.
90 C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 96 HZ<...7!q..i?iü
A0 00 B8 01 02 E8 AA 00 72 19 8B FB B9 0B 00 BE CD .q..ö.r.r.iüf...f=
B0 7D F3 A6 75 0D 8D 7F 20 BE D8 7D B9 0B 00 F3 A6 }<au,iö #+}||..<a
C0 74 18 BE 6E 7D E8 61 00 32 E4 CD 16 5E 1F 8F 04 t.#n)öa.2Z=.^.â.
D0 8F 44 02 CD 19 BE B7 7D EB EB A1 1C 05 33 D2 F7 âD.=.p)ödi..3πz
E0 36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00 6.i.Lö<i7!ü=iq.
F0 07 A1 37 7C E8 40 00 A1 18 7C 2A 06 3B 7C 40 50 .i7!ü0.i.i*.;i0P

```

A:\>

Afb. 16-2. Aan de linkerkant getallen toevoegen.

We komen al dichterbij de volledige afbeelding. Alleen staat onze afbeelding nog niet helemaal gecentreerd op het scherm. We moeten haar ongeveer drie spaties naar rechts verschuiven. We brengen nog even deze laatste verandering aan, en hebben dan de definitieve versie van TOON_REGEL.

We zouden de verandering kunnen aanbrengen door drie keer SCHRIJF_TEK aan te roepen met een spatieteken, maar dat doen we niet. In plaats daarvan zetten we nog een procedure met de naam SCHRIJF_TEK_N_KEER in Video_io. Zoals de naam al aangeeft, schrijft deze procedure een teken N keer. Dat wil zeggen, we zetten het getal N in het CX-register en de code voor het teken in DL, en we roepen SCHRIJF_TEK_N_KEER aan, die dan N kopieën moet schrijven van het teken waarvan we de ASCII-code in DL hebben gezet. Op die manier kunnen we drie spaties schrijven door 3 in CX en 20h (de ASCII-code voor een spatie) in DL te zetten. Dit is de procedure die aan VIDEO_IO.ASM moet worden toegevoegd:

Listing 16-4. Voeg deze procedure toe aan VIDEO_IO.ASM

```

PUBLIC SCHRIJF_TEK_N_KEER
;-----;
; Deze procedure schrijft meer dan een kopie van een teken. ;
; ;
; DL      Code teken ;
; CX      Aantal keren dan teken moet worden geschreven ;
; ;
; Gebruikt:  SCHRIJF_TEK ;
;-----;
SCHRIJF_TEK_N_KEER PROC NEAR
    PUSH    CX
N_KEER:
    CALL    SCHRIJF_TEK
    LOOP    N_KEER
    POP     CX
    RET
SCHRIJF_TEK_N_KEER ENDP

```

U ziet hoe eenvoudig de procedure in elkaar zit, omdat we al over SCHRIJF_TEK beschikken. We hebben ons de moeite getroost om voor zoiets eenvoudigs een procedure te schrijven omdat ons programma Dskpatch veel duidelijker is wanneer we SCHRIJF_TEK_N_KEER aanroepen dan wanneer we een korte lus hadden geschreven die meerdere kopieën van een teken afdrukt. Bovendien zullen we deze procedure nog meermalen kunnen gebruiken.

In TOON_REGEL moeten de volgende veranderingen worden aangebracht om hem drie spaties aan de linkerkant van uw afbeelding te laten toevoegen. Breng de veranderingen in TOON_SEC.ASM aan.

```

PUBLIC TOON_REGEL
EXTRN  SCHRIJF_HEX:NEAR
EXTRN  SCHRIJF_TEK:NEAR
EXTRN  SCHRIJF_TEK_N_KEER:NEAR
;-----;
; Deze procedure drukt een regel met gegevens, of 16 bytes, af; eerst ;
; in hex, daarna in ASCII. ;
; ;
; DS:DX   Offset in sector, in bytes. ;
; ;
; Gebruikt:  SCHRIJF_TEK, SCHRIJF_HEX, SCHRIJF_TEK_N_KEER ;
; Leest:     SECTOR ;
;-----;
TOON_REGEL PROC NEAR
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     BX,DX ;offset is nuttiger in BX
    MOV     DL,' '
    MOV     CX,3 ;schrijf 3 spaties voor regel
    CALL    SCHRIJF_TEK_N_KEER
    ;schrijf offset in hex
    CMP     BX,100h ;is het eerste cijfer een 1?
    JB      SCHRIJF_EEN ;nee, schrijf spatie die al in DL staat
    MOV     DL,'1' ;ja, zet dan '1' in DL voor uitvoer

```

Listing *vervolg*

SCHRIJF_EEN:

.
.
.

We hebben drie dingen veranderd. Eerst moesten we een EXTRN-opdracht voor SCHRIJF__TEK__N__KEER schrijven, omdat de procedure in Video__io staat, en niet in dit bestand. Ook hebben we het commentaarblokje zo veranderd dat het laat zien dat we deze nieuwe procedure gebruiken. Onze derde verandering, de twee regels die SCHRIJF__TEK__N__KEER gebruiken, is vrij eenvoudig en behoeft geen uitleg.

Probeer deze nieuwe versie van ons programma en kijk of de afbeelding nu gecentreerd is. Nu gaan we nog meer aan de afbeelding toevoegen — de bovenste en onderste regels van de kaders.

16.3 Horizontale lijnen toevoegen

Horizontale lijnen aan onze afbeelding toevoegen is niet zo eenvoudig als het klinkt omdat er een paar speciale dingen zijn waaraan we moeten denken. We hebben de einden, waar de lijnen de hoek om moeten, en we hebben de T-vormen boven- en onderaan de scheidingsbalk tussen de hex- en de ASCII-vensters.

We zouden een lange reeks instructies (met SCHRIJF__TEK__N__KEER) kunnen schrijven om onze horizontale lijnen te creëren, maar dat doen we niet. Er is een snellere manier. We maken nog een procedure, genaamd SCHRIJF__PATROON, die een patroon op het scherm schrijft. Het enige wat we daarna nodig hebben, is een klein geheugengebied dat een beschrijving van elk patroon bevat. Met deze nieuwe procedure kunnen we ook gemakkelijk streepjes toevoegen waarmee het hex-venster wordt verdeeld, zoals u aan het einde van deze paragraaf zult zien.

SCHRIJF__PATROON bevat twee volkomen nieuwe instructies, LODSB en CLD. We zullen ze beschrijven nadat we meer hebben gezien van SCHRIJF__PATROON en de manier waarop we een patroon afbeelden. Voer eerst deze procedure in het bestand VIDEO__IO.ASM in:

Listing 16-5. Voeg deze procedure toe aan VIDEO_IO.ASM

```

PUBLIC  SCHRIJF_PATROON
;-----;
; Deze procedure schrijft een regel naar het scherm, op basis van ;
; gegevens in de vorm ;
; ;
; DB      {teken, aantal keren dat teken moet worden geschreven}, 0 ;
; waarbij {x} betekent dat x een willekeurig aantal keren kan worden herhaald.;
; DS:DX   Adres van bovengenoemd datasegment ;
; ;
; Gebruikt:  SCHRIJF_TEK_N_KEER ;
;-----;
SCHRIJF_PATROON PROC    NEAR
    PUSH    AX
    PUSH    CX
```


Listing 16-5. *vervolg*

```

        PUSH    DX
        PUSH    SI
        PUSHF
        CLD
        MOV     SI,DX
PATROON_LUS:
        LODSB
        OR      AL,AL
        JZ      EINDE_PATROON
        MOV     DL,AL
        LODSB
        MOV     CL,AL
        XOR     CH,CH
        CALL    SCHRIJF_TEK_N_KEER
        JMP     PATROON_LUS
EINDE_PATROON:
        POPF
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
SCHRIJF_PATROON ENDP

```

;bewaar richtingvlag
 ;stel richtingvlag in t.b.v ophoging
 ;zet offset in SI-register voor LODSB
 ;haal teken-gegevens naar AL
 ;is 't het einde van gegevens (0h)?
 ;ja, keer terug
 ;nee, klaarmaken voor N keer schrijven
 ;van teken
 ;zet herhaling-teller in AL
 ;en in CX t.b.v SCHRIJF_TEK_N_KEER
 ;hoge byte van CX nul maken
 ;herstel richtingvlag

Voor we kijken hoe deze procedure werkt, bekijken we eerst hoe gegevens voor patronen worden geschreven. We zetten de gegevens voor de bovenste lijn in het bestand `Toon_sec`, waar we ze zullen gebruiken. Daartoe moeten we er nog een procedure met de naam `START_SEC_AFB` aan toevoegen, die de sector-afbeelding opstart door de halve sector-afbeelding te schrijven. Daarna wijzigen we `LEES_SECTOR` zodanig dat hij onze procedure `START_SEC_AFB` aanroept. Zet allereerst de volgende gegevens vlak voor `SECTOR` (in `TOON_SEC.ASM`), binnen het datasegment:

Listing 16-6. Toevoegingen aan `TOON_SEC.ASM`

```

BOVENSTE_REGEL_PATROON LABEL BYTE
    DB      ' ',7
    DB      BOVEN_LINKS,1
    DB      HORIZONTALE_BALK,12
    DB      TOP_STREEPJE,1
    DB      HORIZONTALE_BALK,11
    DB      TOP_STREEPJE,1
    DB      HORIZONTALE_BALK,11
    DB      TOP_STREEPJE,1
    DB      HORIZONTALE_BALK,12
    DB      TOP_T_BALK,1
    DB      HORIZONTALE_BALK,18
    DB      BOVEN_RECHTS,1
    DB      0
ONDERSTE_REGEL_PATROON LABEL BYTE
    DB      ' ',7
    DB      ONDER_LINKS,1

```

Listing 16-6. *vervolg*

```
DB    HORIZONTALE_BALK,12
DB    BENEDEN_STREEPJE,1
DB    HORIZONTALE_BALK,11
DB    BENEDEN_STREEPJE,1
DB    HORIZONTALE_BALK,11
DB    BENEDEN_STREEPJE,1
DB    HORIZONTALE_BALK,12
DB    BENEDEN_T_BALK,1
DB    HORIZONTALE_BALK,18
DB    ONDER_RECHTS,1
DB    0
```

Elke DB-opdracht bevat een deel van de gegevens voor een lijn. De eerste byte is het af te drukken teken; de tweede byte vertelt SCHRIJF_PATROON hoe vaak dat teken moet worden herhaald. We beginnen bijvoorbeeld de bovenste lijn met zeven spaties, gevolgd door een teken voor de linker bovenhoek, gevolgd door twaalf horizontale balk-teken enz. De laatste DB is een enkele hex-nul, die het einde van het patroon aangeeft.

We gaan verder met onze wijzigingen en bekijken het resultaat voor we de interne werking van SCHRIJF_PATROON bespreken. Hieronder staat de testversie van START_SEC_AFB. Deze procedure schrijft het patroon van de bovenste lijn, de halve sector-afbeelding en ten slotte het patroon van de onderste lijn. Zet hem in het bestand TOON_SEC.ASM, vlak voor TOON_HALVE_SECTOR:

Listing 16-7. Voeg deze procedure toe aan TOON_SEC.ASM

```
        PUBLIC  START_SEC_AFB
        EXTRN   SCHRIJF_PATROON:NEAR, STUUR_CRLF:NEAR
;-----;
; Deze procedure start de halve sector-afbeelding. ;
; ;
; Gebruikt:    SCHRIJF_PATROON, STUUR_CRLF, TOON_HALVE_SECTOR ;
; Leest:       BOVENSTE_REGEL_PATROON, ONDERSTE_REGEL_PATROON ;
;-----;
START_SEC_AFB  PROC    NEAR
        PUSH    DX
        LEA     DX,BOVENSTE_REGEL_PATROON
        CALL    SCHRIJF_PATROON
        CALL    STUUR_CRLF
        XOR     DX,DX ;begin bij begin van de sector
        CALL    TOON_HALVE_SECTOR
        LEA     DX,ONDERSTE_REGEL_PATROON
        CALL    SCHRIJF_PATROON
        POP     DX
        RET
START_SEC_AFB  ENDP
```

We hebben de LEA-instructie gebruikt om een adres in het DX-register te laden zodat SCHRIJF_PATROON weet waar hij de patroon-gegevens kan vinden.

Ten slotte moeten we een kleine verandering in LEES_SECTOR in het bestand DISK_IO.ASM aanbrengen zodat START_SEC_AFB in plaats van

TOON__HALVE__SECTOR wordt aangeroepen en er een volledig kader om onze halve sector-afbeelding wordt getekend:

Listing 16-8. Veranderingen in LEES_SECTOR in DISK_IO.ASM

```

        EXTRN  START_SEC_AFB:NEAR
;-----;
; Deze procedure leest de eerste sector op schijf A en dumpt de eerste ;
; helft van deze sector. ;
;-----;
LEES_SECTOR    PROC    NEAR
        MOV     AL,0                ;diskdrive A (nummer 0)
        MOV     CX,1                ;lees maar 1 sector
        MOV     DX,0                ;lees sector 0
        LEA     BX,SECTOR           ;waar deze sector moet worden
                                        ;opgeslagen
        INT     25h                 ;lees de sector
        POPF                                ;verwijder door DOS op stapel
                                        ;gezette vlaggen
        XOR     DX,DX               ;maak offset binnen SECTOR gelijk aan 0
        CALL    START_SEC_AFB       ;dump eerste helft
        INT     20h                 ;terug naar DOS
LEES_SECTOR    ENDP

```

Dat is alles wat er nodig is om de bovenste en onderste lijn van onze sector-afbeelding te schrijven. Assembleer en link al deze bestanden (vergeet niet de drie bestanden die we hebben veranderd, te assembleren), haal het resultaat door Exe2bin en probeer het eens uit. Afb. 16-3 laat de uitvoer zien die we nu hebben:

A:\> disk_io

00	EB 34 90 49 42 4D 20 20 33 2E 32 00 02 02 01 00	64E1BM 3.2....
10	02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00	.p. 1.2.....
20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0F
30	00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 073 14 14 11..
40	BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00	7x.6+7.U.S+!d..
50	FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1	0/80=.t.&e.-e-r+
60	06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10	..eG. 11.+!j=.rga.
70	7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F	!y~&. 11111111u?
80	7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03	!u7!7 ~&. 111111
90	C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 96	H~&. 11111111?i?i?u
A0	00 B8 01 02 E8 AA 00 72 19 8B FB B9 0B 00 BE CD	.. 1111111111111111
B0	7D F3 A6 75 0D 8D 7F 20 BE D8 7D B9 0B 00 F3 A6	}11u. 1111111111111111
C0	74 18 BE 6E 7D E8 61 00 32 E4 CD 16 5E 1F 8F 04	t. 11n)8a.2E=.^..a.
D0	8F 44 02 CD 19 BE B7 7D EB EB A1 1C 05 33 D2 F7	11D.=. 1111111111111111
E0	36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00	6. 1111111111111111
F0	07 A1 37 7C E8 40 00 A1 18 7C 2A 06 3B 7C 40 50	.i7!80.i. 1111111111111111

A:\>

Afb. 16-3. De afbeelding met gesloten kaders.

Laten we eens kijken hoe SCHRIJF__PATROON werkt. Zoals gezegd, gebruikt hij twee nieuwe instructies. LODSB staat voor *Load String Byte* en is een van de *string-instructies* — speciaal ontworpen instructies die met tekenstrings werken. Dat is niet helemaal wat we hier doen, maar het kan de 8088 niet schelen of we met een tekenstring of alleen getallen werken, dus LODSB voorziet prima in onze behoefte.

LODSB zet (laadt) een enkele byte in het AL-register vanuit de geheugenplaats die is gegeven in DS:SI, een registerpaar dat we nog niet eerder hebben gebruikt. Alle segmentregisters in ons .COM-bestand worden ingesteld op het begin van ons ene segment, CGROEP, dus DS is al op ons segment ingesteld. En voor de LODSB-instructie hebben we met de instructie MOV SI,DX de offset in het SI-register gezet. De LODSB-instructie heeft wel iets weg van de MOV-instructie, maar is krachtiger. Met een LODSB-instructie zet de 8088 een byte in het AL-register en hoogt dan het SI-register op of verlaagt het. Bij ophogen wijst het SI-register naar de volgende byte in het geheugen, bij verlagen naar de vorige byte in het geheugen.

Dat eerste (ophogen) is precies wat we willen. We willen het hele patroon doornemen, met één byte tegelijk, vanaf het begin, en dat doet de LODSB-instructie voor ons, omdat we de andere nieuwe instructie, CLD (*Clear Direction Flag*) hebben gebruikt om de richtingvlag op nul te zetten. Als we de richtingvlag op 1 hadden gezet, zou de LODSB-instructie het SI-register in plaats daarvan verlagen. We zullen de LODSB-instructie op nog enkele andere plaatsen in Dskpatch gebruiken, steeds met de richtingvlag op 0 gezet, om op te hogen.

Afgezien van LODSB en CLD hebt u misschien gezien dat we ook de PUSHF- en POPF-instructie hebben gebruikt om het vlaggenreger te bewaren en te herstellen. We hebben dat gedaan voor het geval we later besluiten de richtingvlag te gebruiken in een procedure die SCHRIJF__PATROON aanroept.

16.4 Getallen aan de afbeelding toevoegen

We zijn nu bijna klaar met deel 2 van dit boek. We schrijven nog één procedure, en gaan dan door naar deel 3, waar ons grotere en betere dingen wachten.

Maar kijk eerst even naar de bovenkant van onze afbeelding, waar een rij getallen ontbreekt. Met zulke getallen — 00 01 02 03, enz. — zouden we beter de kolommen met de adressen van elke byte kunnen vinden. We gaan daarom een procedure schrijven die deze reeks getallen afdrukt. Zet deze procedure, SCHRIJF__HEX__GETALLEN__BOVENLANGS, in TOON__SEC.ASM, vlak na START__SEC__AFB:

Listing 16-9. Zet deze procedure in TOON__SEC.ASM

```

        EXTRN    SCHRIJF_TEK_N_KEER:NEAR, SCHRIJF_HEX:NEAR, SCHRIJF_TEK:NEAR
        EXTRN    SCHRIJF_HEX_CIJFER:NEAR, STUUR_CRLF:NEAR
;-----;
; Deze procedure schrijft de getallen 0 t/m F bovenaan de halve ;
; sector-afbeelding ;
; ;
; Gebruikt:      SCHRIJF_TEK_N_KEER, SCHRIJF_HEX, SCHRIJF_TEK ;
;               SCHRIJF_HEX_CIJFER, STUUR_CRLF ;
;-----;
SCHRIJF_HEX_GETALLEN_BOVENLANGS PROC    NEAR
        PUSH    CX

```


Listing 16-9. *vervolg*

```

        PUSH    DX
        MOV     DL,' '           ;schrijf 9 spaties voor linkerkant
        MOV     CX,9
        CALL    SCHRIJF_TEK_N_KEER
        XOR     DH,DH           ;begin met 0
HEX_GETAL_LUS:
        MOV     DL,DH
        CALL    SCHRIJF_HEX
        MOV     DL,' '
        CALL    SCHRIJF_TEK
        INC     DH
        CMP     DH,10h          ;klaar?
        JB      HEX_GETAL_LUS

        MOV     DL,' '           ;schrijf hex-getallen boven ASCII-venster
        MOV     CX,2
        CALL    SCHRIJF_TEK_N_KEER
        XOR     DL,DL
HEX_CIJFER_LUS:
        CALL    SCHRIJF_HEX_CIJFER
        INC     DL
        CMP     DL,10h
        JB      HEX_CIJFER_LUS
        CALL    STUUR_CRLF
        POP     DX
        POP     CX
        RET
SCHRIJF_HEX_GETALLEN_BOVENLANGS ENDP

```

Wijzig `START_SEC_AFB` (ook in `TOON_SEC.ASM`) als volgt, zodat hij `SCHRIJF_HEX_GETALLEN_BOVENLANGS` aanroept vóór hij de rest van de halve sector-afbeelding schrijft:

Listing 16-10. Veranderingen van `START_SEC_AFB` in `TOON_SEC.ASM`

```

;-----;
; Deze procedure start de halve sector-afbeelding op. ;
; ;
; Gebruikt:  SCHRIJF_PATROON, STUUR_CRLF, TOON_HALVE_SECTOR ;
;           SCHRIJF_HEX_GETALLEN_BOVENLANGS ;
; Leest:  "  BOVENSTE_REGEL_PATROON, ONDERSTE_REGEL_PATROON ;
;-----;
START_SEC_AFB  PROC    NEAR
        PUSH    DX
        CALL    SCHRIJF_HEX_GETALLEN_BOVENLANGS
        LEA     DX,BOVENSTE_REGEL_PATROON
        CALL    SCHRIJF_PATROON
        CALL    STUUR_CRLF
        XOR     DX,DX           ;begin bij begin van de sector
        CALL    TOON_HALVE_SECTOR
        LEA     DX,ONDERSTE_REGEL_PATROON
        CALL    SCHRIJF_PATROON
        POP     DX
        RET
START_SEC_AFB  ENDP

```

Nu hebben we een volledige halve sector-afbeelding, zoals u in afb. 16-4 kunt zien:

A:\> disk io

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	34	90	49	42	4D	20	20	33	2E	32	00	02	02	01	00	64EIBM 3.2....
10	02	70	00	D0	02	FD	02	00	09	00	02	00	00	00	00	00	.p.u.2.....
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0F3LHJ...i..
30	00	00	00	00	01	00	FA	33	C0	8E	D0	BC	00	7C	16	07x.6+7.U.S+id..
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	ny&=,t.&e.-e-r+
50	FC	AC	26	80	3D	00	74	03	26	8A	05	AA	8A	C4	E2	F1	..eG. .+i =,rga,
60	06	1F	89	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	..y&.....u?
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	..u7i7..&..i.i?
80	7C	A3	37	7C	B8	20	00	F7	26	11	7C	8B	1E	0B	7C	03	H~...7i7..i?i8u
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	96	..f...8r.r.i7j...3=
A0	00	B8	01	02	E8	AA	00	72	19	8B	FB	B9	0B	00	BE	CD	}_au.id #7j...<a
B0	7D	F3	A6	75	0D	8D	7F	20	BE	D8	7D	B9	0B	00	F3	A6	t,3n>8a,28=,^,A,
C0	74	18	BE	6E	7D	E8	61	00	32	E4	CD	16	5E	1F	8F	04	AD,=,3p>88i...3m~
D0	8F	44	02	CD	19	BE	B7	7D	EB	EB	A1	1C	05	33	D2	F7	6..iL<i7i8=i7,
E0	36	0B	7C	FE	C0	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	.i7i8e.i.i*.;i8P
F0	07	A1	37	7C	E8	40	00	A1	18	7C	2A	06	3B	7C	40	50	

A:\>

Afb. 16-4. Een volledige halve sector-afbeelding.

Er bestaan nog enkele verschillen tussen deze afbeelding en de uiteindelijke versie. We zullen SCHRIJF__TEK nog zo veranderen dat hij alle 256 tekens afdruckt die de IBM-PC kan laten zien, en daarna maken we het scherm schoon en centreren de afbeelding verticaal met behulp van de ROM BIOS-routines in de IBM Personal Computer. Dat doen we hierna.

16.5 Samenvatting

We hebben nogal wat geknutseld aan ons Dskpatch-programma, er nieuwe procedures aan toegevoegd en oude veranderd, en we zijn van het ene bronbestand naar het andere overgegaan. Als u van nu af niet meer kunt bijhouden wat u doet, bekijk dan de volledige listing van Dskpatch in bijlage B. De listing daar is de uiteindelijke versie, maar u ziet vermoedelijk voldoende overeenkomst om u op weg te helpen.

De meeste veranderingen die we in dit hoofdstuk hebben aangebracht, zijn niet op trucs gebaseerd maar op gewoon hard werken. Maar we hebben wel twee nieuwe instructies geleerd: LODSB en CLD. LODSB is een van de stringinstructies waarmee met één instructie het werk van meerdere kunnen doen. We hebben LODSB in SCHRIJF__PATROON gebruikt om achtereenvolgende bytes uit de patroon-tabel te halen, en daarbij steeds een nieuwe byte in het AL-register gezet. CLD zet de richtingvlag op nul, waardoor de richting op ophogen wordt gezet. Elke volgende LODSB-instructie laadt de volgende byte uit het geheugen.

In het volgende deel van dit boek leren we over de ROM BIOS-routines. Die zullen ons een hoop tijd besparen.

Deel 3

Het ROM BIOS van de IBM PC

17 De ROM BIOS-routines

- 17.1 VIDEO_IO, de ROM BIOS-routines 188
- 17.2 Verplaatsen van de cursor 193
- 17.3 Ander gebruik van variabelen 194
- 17.4 De kopregel schrijven 198
- 17.5 Samenvatting 200

Binnenin uw IBM Personal Computer zitten enkele chips, of IC's (*Integrated Circuits*, geïntegreerde schakelingen) die bekend staan als ROM (*Read-Only Memory*, onuitwisbaar geheugen). Eén van deze ROMs bevat routines die erg veel weg hebben van procedures en alle elementaire werkzaamheden verrichten in verband met in- en uitvoer voor verschillende delen van de IBM-PC. Omdat dit ROM routines verschaft voor het verrichten van in- en uitvoer op een zeer laag niveau, worden ze vaak aangeduid met de term BIOS (*Basic Input Output System*). DOS gebruikt het ROM BIOS voor activiteiten als het sturen van tekens naar het scherm en het lezen en schrijven van en naar de schijf, en we kunnen de ROM BIOS-routines in onze programma's naar believen gebruiken.

We zullen ons richten op de BIOS-routines die we nodig hebben voor Dskpatch. Daaronder valt een aantal routines voor afbeeldingen op het scherm, met enkele functies die we normaliter niet zouden kunnen aanspreken zonder rechtstreeks met de hardware te werken — een zeer moeilijk klusje.

17.1 VIDEO__IO, de ROM BIOS-routines

We noemen de onderdelen van het ROM BIOS-routines om ze te onderscheiden van procedures. We gebruiken procedures met behulp van CALL-instructies, terwijl we routines met INT-instructies aanroepen, niet met CALLs. Zo gebruiken we een instructie INT 10h voor het aanroepen van de videoroutines voor in- en uitvoer naar het scherm, net zoals we een INT 21h-instructie gebruikten om routines in DOS aan te roepen.

INT 10h roept de routine VIDEO__IO in het ROM BIOS op. Andere nummers roepen andere routines aan, maar daar zullen we er geen van zien; VIDEO__IO verschaft alle functies die we buiten DOS nodig hebben. (DOS roept echter een van de andere ROM BIOS-routines aan wanneer we om een sector van de schijf vragen.) In dit hoofdstuk zullen we ROM BIOS-routines gebruiken om twee nieuwe procedures aan Dskpatch toe te voegen: een om het scherm schoon te maken en de andere om de cursor op elke plaats op het scherm te zetten die we willen. Het zijn twee nuttige functies, maar ze zijn geen van beide rechtstreeks via DOS beschikbaar. Daarom zullen we daar de ROM BIOS-routines voor gebruiken. Verderop zullen we nog interessantere dingen zien die we met deze ROM-routines kunnen doen, maar laten we eerst INT 10h gebruiken om het scherm te wissen voor we onze halve sector afbeelden.

De instructie INT 10h biedt ons toegang tot een aantal verschillende functies. U zult nog weten dat, toen we de DOS-instructie INT 21h gebruiken, we een bepaalde functie selecteerden door het functienummer in het AH-register te zetten. Op precies dezelfde manier selecteren we een VIDEO__IO-functie: door het desbetreffende functienummer in het AH-register te zetten. Een volledige lijst van deze functies staat in de nu volgende tabel:

<i>INT 10h-Functies</i>	
Functienummer	Omschrijving
(AH)=0	Videomodus instellen. Het AL-register bevat het modusnummer. <p style="text-align: center;"><i>Tekstmodi</i></p> (AL)=0 40 bij 25, zwart-wit (AL)=1 40 bij 25, kleur (AL)=2 80 bij 25, zwart-wit (AL)=3 80 bij 25, kleur (AL)=7 80 bij 25, monochrome video-adapter <p style="text-align: center;"><i>Grafische modus</i></p> (AL)=4 320 bij 200, kleur (AL)=5 320 bij 200, zwart-wit (AL)=6 640 bij 200, zwart-wit
(AH)=1	Cursorgrootte instellen. <p>(CH) Eerste beeldlijn van de cursor. De bovenste lijn is 0 op zowel de monochrome als kleur/grafische schermen, terwijl de onderste lijn 7 is voor de kleur/grafische adapter en 13 voor de monochrome adapter. Geldig bereik: 0 t/m 31.</p> <p>(CL) Laatste beeldlijn van de cursor.</p> <p>De instelling bij aanzetten bij de kleur/grafische adapter is CH = 6 en CL = 7. Voor het monochrome scherm: CH = 11 en CL = 12.</p>
(AH)=2	Cursorpositie instellen. <p>(DH,DL) Rij, kolom van nieuwe cursorpositie; de linker bovenhoek is (0,0).</p> <p>(BH) Paginanummer. Dit is het nummer van de scherpagina. De kleur/grafische adapter heeft plaats voor meerdere scherpagina's, maar de meeste programma's gebruiken pagina 0.</p>
(AH)=3	Cursorpositie lezen. <p>(BH) Paginanummer</p> <p>retour: (DH,DL) Rij, kolom van cursor (CH,CL) Cursorgrootte</p>

<i>INT 10h-Functies</i>	
Functienummer	Omschrijving
(AH)=4	Positie lichtpen lezen (zie <i>Technical Reference</i> -handleiding).
(AH)=5	Actieve scherpagina selecteren.
(AL)	Nieuwe paginanummer (van 0 t/m 7 voor modi 0 en 1; van 0 t/m 3 voor modi 2 en 3)
(AH)=6	Pagina omhoog schuiven.
(AL)	Aantal leeg te maken regels onderaan op het scherm. Bij normaal verschuiven wordt er een regel gewist. Stel op 0 in om hele venster leeg te maken.
(CH,CL)	Rij, kolom van linker bovenhoek van venster
(DH,DL)	Rij, kolom van rechter benedenhoek van venster
(BH)	Schermattribuut voor blanco regels
(AH)=7	Pagina omlaag schuiven.
	Zelfde als pagina omhoog schuiven (functie 6), maar regels blijven bovenaan op het scherm leeg in plaats van onderaan.
(AH)=8	Lees attribuut en teken onder de cursor.
(BH)	Schermpagina (alleen tekstmodi)
(AL)	Gelezen teken
(AH)	Attribuut van gelezen teken (alleen tekstmodi)
(AH)=9	Schrijf attribuut en teken onder de cursor.
(BH)	Schermpagina (alleen tekstmodi)
(CX)	Aantal keren dat teken en attribuut naar scherm moeten worden geschreven
(AL)	Te schrijven teken
(BL)	Te schrijven attribuut
(AH)=10	Schrijf teken onder cursor (bij normaal attribuut).
(BH)	Schermpagina
(CX)	Aantal keren dat teken moet worden geschreven
(AL)	Te schrijven teken

INT 10h-Functies	
Functienummer	Omschrijving
(AH)=11 t/m 13	Verschillende grafische functies. (Zie <i>Technical Reference-handleiding</i> voor nadere bijzonderheden.)
(AH)=14	Schrijf telex. Schrijf een teken naar het scherm en zet de cursor naar de volgende plaats. (AL) Te schrijven teken (BL) Kleur van teken (alleen grafische modus) (BH) Schermpagina (tekstmodus)
(AH)=15	Huidige videostatus doorgeven. (AL) Toon huidige modus (AH) Aantal tekens per regel (BH) Actieve schermpagina's

We zullen de INT 10h-functie 6, Pagina omhoog schuiven, gebruiken om het scherm leeg te maken. We willen het scherm niet echt verschuiven, maar deze functie kan ook dienen om het scherm te wissen. De procedure is als volgt; zet hem in het bestand CURSOR.ASM

Listing 17-1. Voeg deze procedure toe aan CURSOR.ASM

```

PUBLIC WIS_SCHERM
;-----;
; Deze procedure wist het hele scherm.
;-----;
WIS_SCHERM PROC NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    XOR     AL,AL      ;maak hele venster leeg
    XOR     CX,CX      ;linker bovenhoek is (0,0)
    MOV     DH,24      ;onderste regel van scherm is regel 24
    MOV     DL,79      ;rechterkant is kolom 79
    MOV     BH,7       ;normale attribuut voor lege regels
    MOV     AH,6       ;roep functie voor omhoog schuiven aan
    INT     10h        ;maak venster leeg
    POP     DX
    POP     CX

```


Listing 17-1. *vervolg*

```

        POP     BX
        POP     AX
        RET
WIS_SCHERM      ENDP

```

Functie 6 van INT 10h lijkt nogal wat informatie nodig te hebben, al willen we alleen het scherm leegmaken. Deze functie is vrij krachtig: ze kan eigenlijk elke willekeurige rechthoek van het scherm — *venster*, zoals we dat noemen — leegmaken. We moeten het venster definiëren als het hele scherm door de eerste en laatste regel op 0 en 24 te zetten, en de kolommen op 0 en 79. De routines die we hier gebruiken, kunnen het scherm ook helemaal wit maken (voor gebruik met zwarte tekens), of helemaal zwart (voor gebruik met witte tekens). Het laatste is onze bedoeling, en dat hebben we ook aangegeven met de instructie MOV BH,7. Door AL op 0, het aantal te verschuiven regels, te zetten, krijgt de routine hier de opdracht het venster te wissen in plaats van het te verschuiven.

Nu moeten we onze testprocedure LEES_SECTOR zo wijzigen dat hij WIS_SCHERM aanroept vlak voor hij naar de sector-afbeelding begint te schrijven. We hebben deze aanroep niet in START_SEC_AFB gezet omdat we willen dat START_SEC_AFB alleen de halve sector-afbeelding herschrijft zonder de rest van het scherm te beïnvloeden.

Om LEES_SECTOR te wijzigen moet u een EXTRN-declaratie voor WIS_SCHERM toevoegen en de aanroep van WIS_SCHERM invoegen. Breng de volgende wijzigingen aan in DISK_IO.ASM:

Listing 17-2. Wijzigingen van LEES_SECTOR in DISK_IO.ASM

```

        EXTRN  START_SEC_AFB:NEAR, WIS_SCHERM:NEAR
;-----;
; Deze procedure leest de eerst sector op schijf A en dumpt de eerste ;
; helft van deze sector. ;
;-----;
LEES_SECTOR  PROC    NEAR
        MOV     AL,0          ;diskdrive A (nummer 0)
        MOV     CX,1          ;lees maar 1 sector
        MOV     DX,0          ;lees sector 0
        LEA     BX,SECTOR     ;waar deze sector moet worden
                                ;opgeslagen
        INT     25h           ;lees de sector
        POPF                    ;verwijder door DOS op stapel
                                ;gezette vlaggen
        XOR     DX,DX         ;maak offset binnen SECTOR gelijk aan 0
        CALL    WIS_SCHERM
        CALL    START_SEC_AFB ;dump eerste helft
        INT     20h           ;terug naar DOS
LEES_SECTOR  ENDP

```

Merk op waar de cursor staat vlak voor u de nieuwe versie van Disk_io draait. Draai dan Disk_io. Het scherm wordt leeggemaakt, en Disk_io zal de halve sector-afbeelding beginnen te schrijven vanaf de plaats waar de cursor stond voor u het programma draaide — waarschijnlijk onderaan op het scherm.

We hebben nu het scherm gewist, maar we hebben het nog niet gehad over het weer verplaatsen van de cursor naar de bovenkant. In BASIC maakt de CLS-opdracht het scherm in twee stappen schoon: hij wist het scherm en zet de cursor naar de bovenkant. Dat doet onze procedure niet; we zullen de cursor zelf moet verplaatsen.

17.2 Verplaatsen van de cursor

Functie 2 van INT 10h stelt de cursorpositie haast net zo in als de LOCATE-opdracht in BASIC. We kunnen GANAAR_XY gebruiken om de cursor overal op het scherm te zetten (zoals links boven na het wissen), maar dat doen we niet. Voer deze procedure in het bestand CURSOR.ASM IN:

Listing 17-3. Zet deze procedure in CURSOR.ASM

```

PUBLIC GANAAR_XY
;-----;
; Deze procedure verplaatst de cursor.
;
; DH      Rij (Y)
; DL      Kolom (X)
;-----;
GANAAR_XY PROC NEAR
    PUSH    AX
    PUSH    BX
    MOV     BH,0           ;schermpagina 0
    MOV     AH,2           ;aanroep Cursorpositie instellen
    INT     10h
    POP     BX
    POP     AX
    RET
GANAAR_XY ENDP

```

We gebruiken GANAAR_XY in een herziene versie van START_SEC_AFB, om de cursor vlak voor we de halve sector-afbeelding schrijven naar de tweede regel te verplaatsen. Dit zijn de wijzigingen die daarvoor nodig zijn in START_SEC_AFB in TOON_SEC.ASM:

Listing 17-4. Veranderingen van START_SEC_AFB in TOON_SEC.ASM

```

PUBLIC START_SEC_AFB
EXTRN SCHRIJF_PATROON:NEAR, STUUR_CRLF:NEAR
EXTRN GANAAR_XY:NEAR
;-----;
; Deze procedure start de halve sector-afbeelding.
;
; Gebruikt:  SCHRIJF_PATROON, STUUR_CRLF, TOON_HALVE_SECTOR
;            SCHRIJF_HEX_GETALLEN_BOVENLANGS, GANAAR_XY
; Leest:     BOVENSTE_REGEL_PATROON, ONDERSTE_REGEL_PATROON
;-----;
START_SEC_AFB PROC NEAR
    PUSH    DX
    XOR     DL,DL           ;zet cursor op plaats aan begin
    MOV     DH,2           ; van 3de regel

```


Listing 17-4. *vervolg*

```
CALL    GANAAR_XY
CALL    SCHRIJF_HEX_GETALLEN_BOVENLANGS
LEA     DX,BOVENSTE_REGEL_PATROON
CALL    SCHRIJF_PATROON
CALL    STUUR_CRLF
XOR     DX,DX                      ;begin bij begin van de sector
CALL    TOON_HALVE_SECTOR
        .
        .
        .
```

Als u het nu probeert, zult u zien dat de afbeelding mooi gecentreerd staat. Zoals u nu ziet, is het gemakkelijk om met het scherm te werken als we de ROM BIOS-routines gebruiken. In het volgende hoofdstuk zullen we nog een routine in het ROM BIOS gebruiken om SCHRIJF__TEK zo te veranderen dat hij elk teken naar het scherm schrijft. Maar laten we voor we verder gaan nog wat in ons programma veranderen, en dan eindigen met een procedure genaamd SCHRIJF__KOP die een statusregel aan de bovenkant van het scherm schrijft met vermelding van de huidige diskdrive en sectornummer.

17.3 Ander gebruik van variabelen

Er is nog veel op te knappen voor we SCHRIJF__KOP schrijven. In de huidige vorm werken veel van onze procedures met vaste getallen; LEES__SECTOR leest bijvoorbeeld sector 0 van drive A. Wat we willen is de nummers van de diskdrive en de sector in geheugenvariabelen zetten zodat ze door meerdere procedures kunnen worden gelezen.

We zullen deze procedures zo moeten veranderen dat ze geheugenvariabelen gebruiken, maar laten we beginnen met alle geheugenvariabelen in één bestand, DSKPATCH.ASM, te zetten om het ons gemakkelijker te maken. Dskpatch.asm wordt het eerste bestand in ons programma Dskpatch, dus de geheugenvariabelen zullen daar gemakkelijk te vinden zijn. Hier is DSKPATCH.ASM, compleet met een lange lijst met variabelen:

Listing 17-5 Het nieuwe bestand DSKPATCH.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG  SEGMENT PUBLIC
        ORG    100h

        EXTRN WIS_SCHERM:NEAR, LEES_SECTOR:NEAR
        EXTRN START_SEC_AFB:NEAR

DISK_PATCH  PROC    NEAR
        CALL    WIS_SCHERM
        CALL    LEES_SECTOR
        CALL    START_SEC_AFB
```

Listing 17-5. *vervolg*

```

        INT      20h
DISK_PATCH  ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC

        PUBLIC  SECTOR_OFFSET
;-----;
; SECTOR_OFFSET is de offset van de halve
; sector-afbeelding in de volledige sector. Hij
; moet een veelvoud van 16 zijn, en niet groter
; dan 256.
;-----;
SECTOR_OFFSET  DW      0

        PUBLIC  HUIDIGE_SECTORNR, DISKDRIVE_NR
HUIDIGE_SECTORNR  DW      0          ;aanvankelijk sector 0
DISKDRIVE_NR      DB      0          ;aanvankelijk drive A:

        PUBLIC  REGELS_VOOR_SECTOR, KOPREGEL_NR
        PUBLIC  KOP_DEEL_1, KOP_DEEL_2
;-----;
; REGELS_VOOR_SECTOR is het aantal regels
; bovenaan op het scherm voor de halve sector-
; afbeelding.
;-----;
REGELS_VOOR_SECTOR  DB      2
KOPREGEL_NR         DB      0
KOP_DEEL_1          DB      'Drive ',0
KOP_DEEL_2          DB      '      Sector ',0

        PUBLIC  SECTOR
;-----;
; De hele sector (tot aan 8192 bytes) is
; opgeslagen in dit deel van het geheugen.
;-----;
SECTOR  DB      8192 DUP (0)

DATA_SEG  ENDS

        END      DISK_PATCH

```

De hoofdprocedure, DISK_PATCH, roept drie andere procedures aan. We hebben ze alle drie al gezien; straks zullen we zowel LEES_SECTOR als START_SEC_AFB zo veranderen dat ze de variabelen gebruiken die zojuist in het datasegment zijn gezet.

Voor we Dskpatch kunnen gebruiken, moeten we de definitie van SECTOR in Toon_sec vervangen door een EXTRN. Ook moeten we Disk_io wijzigen en LEES_SECTOR veranderen in een gewone procedure die we vanuit Dskpatch kunnen aanroepen.

We nemen eerst SECTOR. Omdat we die als geheugenvariabele in DSKPATCH.ASM hebben gezet, moeten we de definitie van SECTOR in Toon_sec zo veranderen dat

het een EXTRN-declaratie wordt. Breng deze wijzigingen in TOON_SEC.ASM aan:

Listing 17-6 Veranderingen in TOON_SEC.ASM

```
DATA_SEG          SEGMENT PUBLIC
                  EXTRN  SECTOR:BYTE
                  PUBLIC SECTOR
SECTOR DB         512 DUP(0)

BOVENSTE_REGEL_PATROON LABEL BYTE
DB      ' ',7
DB      BOVEN_LINKS,1
      .
      .
      .
```

Nu wijzigen we het bestand DISK_IO.ASM zo dat het alleen nog maar procedures bevat, en LEES_SECTOR geheugenvariabelen (geen vaste getallen) gebruikt voor de nummers van de sectoren en diskdrives gebruikt. Dit is de nieuwe versie van DISK_IO.ASM:

Listing 17-7. Veranderingen in DISK_IO.ASM

```
CGROEP GROUP CODE_SEG, DATA_SEG
      ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG          SEGMENT PUBLIC
      ORG         100h
      PUBLIC      LEES_SECTOR
DATA_SEG          SEGMENT PUBLIC
      EXTRN       SECTOR:BYTE
      EXTRN       DISKDRIVE_NR:BYTE
      EXTRN       HUIDIGE_SECTORNR:WORD
DATA_SEG          ENDS
      EXTRN       START_SEC_AFB:NEAR, WIS_SCHERM:NEAR

;-----;
; Deze procedure leest een sector (512 bytes) naar SECTOR. ;
; ;
; Leest:          HUIDIGE_SECTORNR, DISKDRIVE_NR ;
; Schrijft:       SECTOR ;
;-----;
LEES_SECTOR PROC NEAR
      PUSH AX
      PUSH BX
      PUSH CX
      PUSH DX
      MOV AL,DISKDRIVE_NR ;drive-nummer
      MOV CX,1 ;lees maar 1 sector
      MOV DX,HUIDIGE_SECTORNR ;logische sector-nummer
      MOV AL,0 ;diskdrive A (nummer 0)
      MOV CX,1 ;lees maar een 1 sector
      MOV DX,0 ;lees sector 0
      LEA BX,SECTOR ;waar deze sector moet worden
                  ; opgeslagen
      INT 25h ;lees de sector
```

Listing 17-7. *vervolg*

```

        POPF                ;verwijder door DOS op stapel
                                ; gezette vlaggen
        XOR     DX,DX        ;maak offset binnen SECTOR gelijk aan 0
        CALL    WIS_SCHERM
        CALL    START_SEC_AFB ;dump eerste helft
        INT     20h         ;terug naar DOS
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LEES_SECTOR    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
               EXTRN    SECTOR:BYTE
DATA_SEG      ENDS
END

```

Deze nieuwe versie van Disk_io gebruikt de geheugenvariabelen DISKDRIVE__NR en HUIDIGE__SECTORNR als nummers van de diskdrive en sector voor de te lezen sector. Omdat deze variabelen al in DSKPATCH.ASM zijn gedefinieerd, hoeven we Disk_io niet te wijzigen wanneer we verschillende sectoren van andere diskdrives beginnen te lezen.

Als u het programma Make gebruikt om DSKPATCH.COM om te bouwen, moet u een aantal dingen aan het Make-bestand genaamd Dskpatch toevoegen:

Listing 17-8. De nieuwe versie van DSKPATCH

```

dskpatch.obj:  dskpatch.asm
               masm dskpatch;

disk_io.obj:   disk_io.asm
               masm disk_io;

toon_sec.obj:  toon_sec.asm
               masm toon_sec;

video_io.obj:  video_io.asm
               masm video_io;

cursor.obj:    cursor.asm
               masm cursor;

dskpatch.com:  dskpatch.obj disk_io.obj toon_sec.obj video_io.obj cursor.obj
               link dskpatch disk_io toon_sec video_io cursor;
               exe2bin dskpatch dskpatch.com

```

Als u Make niet gebruikt, assembleer dan alle drie bestanden die u hebt veranderd (Dskpatch, Disk_io en Toon_sec) en link alle vijf bestanden, met Dskpatch voor-


```
LINK DSKPATCH DISK_IO TOON_SEC VIDEO_IO CURSOR;
EXE2BIN DSKPATCH DSKPATCH.COM
```

We hebben nogal wat veranderingen aangebracht, dus test Dskpatch en zorg dat het goed werkt voor u verder gaat.

17.4 De kopregel schrijven

Nu we de vaste getallen hebben omgezet in de rechtstreekse verwijzingen naar geheugenvariabelen, kunnen we de procedure `SCHRIJF_KOP` schrijven om de statusregel bovenaan op het scherm te kunnen afbeelden. Onze kop zal er zo uitzien:

Drive A Sector 0

`SCHRIJF_KOP` zal gebruik maken van `SCHRIJF_DECIMAAL` om het nummer van de huidige sector in decimaal uit te schrijven. Hij zal ook twee tekenstrings, *Drive* en *Sector* schrijven, elk gevolgd door een spatie, en een drive-letter, zoals A. We zetten de procedure in het bestand `VIDEO_IO.ASM`.

Omdat we een verwijzing naar het datasegment (`DATA_SEG`) hebben, veranderen we allereerst de eerste regel (de `GROUP`-opdracht) in `VIDEO_IO.ASM` zodanig dat hij er zo uitziet:

```
CGROEP GROUP CODE_SEG, DATA_SEG
```

Zet nu de volgende procedure in `VIDEO_IO.ASM`:

Listing 17-9. Voeg deze procedure toe aan `VIDEO_IO.ASM`

```

PUBLIC SCHRIJF_KOP
DATA_SEG SEGMENT PUBLIC
    EXTRN KOPREGEL_NR:BYTE
    EXTRN KOP_DEEL_1:BYTE
    EXTRN KOP_DEEL_2:BYTE
    EXTRN DISKDRIVE_NR:BYTE
    EXTRN HUIDIGE_SECTORNR:WORD
DATA_SEG ENDS
    EXTRN GANAAR_XY:NEAR
;-----;
; Deze procedure schrijft de kop met diskdrive en sector-nummer. ;
; ;
; Gebruikt: GANAAR_XY, SCHRIJF_STRING, SCHRIJF_TEK, SCHRIJF_DECIMAAL;
; Leest: KOPREGEL_NR, KOP_DEEL_1, KOP_DEEL_2 ;
; DISKDRIVE_NR, HUIDIGE_SECTORNR ;
;-----;
SCHRIJF_KOP PROC NEAR
    PUSH DX
    XOR DL,DL ;zet cursor op regelnummer kop
    MOV DH,KOPREGEL_NR
    CALL GANAAR_XY
    LEA DX,KOP_DEEL_1
    CALL SCHRIJF_STRING
    MOV DL,DISKDRIVE_NR
    ADD DL,'A' ;druk drives A, B, ... af

```

Listing 17-9. *vervolg*

```

CALL    SCHRIJF_TEK
LEA     DX,KOP_DEEL_2
CALL    SCHRIJF_STRING
MOV     DX,HUIDIGE_SECTORNR
CALL    SCHRIJF_DECIMAAL
POP     DX
RET
SCHRIJF_KOP    ENDP

```

De procedure `SCHRIJF_STRING` bestaat nog niet. Zoals u ziet, willen we hem gebruiken om een tekenstring naar het scherm te schrijven. De twee strings, `KOP_DEEL_1` en `KOP_DEEL_2`, zijn al gedefinieerd in `DSKPATCH.ASM`. `SCHRIJF_STRING` zal `DS:DX` gebruiken als adres voor de string.

We hebben gekozen voor een eigen stringuitvoer-procedure om onze strings elk teken te kunnen laten bevatten, met inbegrip van de \$, die we niet met de DOS-functie 9 zouden kunnen afdrukken. Waar DOS een \$ gebruikt om het einde van een string aan te geven, zullen wij hex 0 gebruiken. Voer het in `VIDEO_IO.ASM` in.

Listing 17-10. Voeg deze procedure toe aan `VIDEO_IO.ASM`

```

PUBLIC  SCHRIJF_STRING
;-----;
; Deze procedure schrijft een tekenstring naar het scherm. De string ;
; moet eindigen op          DB      0 ;
; ; ;
; DS:DX  Adres van de string ;
; ; ;
; Gebruikt:  SCHRIJF_TEK ;
;-----;
SCHRIJF_STRING PROC NEAR
    PUSH    AX
    PUSH    DX
    PUSH    SI
    PUSHF   ;bewaar richtingvlag
    CLD     ;stel richting in op ophogen
            ;(naar voren)
    MOV     SI,DX ;zet adres in SI voor LODSB
STRING_LUS:
    LODSB   ;zet een teken in het AL-register
    OR      AL,AL ;hebben we de 0 al gevonden?
    JZ      EINDE_STRING ;ja, klaar met de string
    MOV     DL,AL ;nee, schrijf teken
    CALL    SCHRIJF_TEK
    JMP     STRING_LUS
EINDE_STRING:
    POPF    ;herstel richtingvlag
    POP     SI
    POP     DX
    POP     AX
    RET
SCHRIJF_STRING ENDP

```


In deze vorm zal SCHRIJF_STRING tekens met ASCII-codes beneden de 32 (de spatie) als een punt uitschrijven omdat we geen versie van SCHRIJF_TEK hebben die *elk* teken schrijft. Daar zorgen we in het volgende hoofdstuk voor.

Laten we na al ons werk in dit hoofdstuk er eens de kroon op zetten. Wijzig DISK_PATCH in DSKPATCH.ASM zodanig dat hij nu ook de aanroep van SCHRIJF_KOP bevat:

Listing 17-11. Veranderingen aan DISK_PATCH in DSKPATCH.ASM

```

EXTRN    WIS_SCHERM:NEAR, LEES_SECTOR:NEAR
EXTRN    START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
DISK_PATCH    PROC    NEAR
CALL      WIS_SCHERM
CALL      SCHRIJF_KOP
CALL      LEES_SECTOR
CALL      START_SEC_AFB
INT       20h
DISK_PATCH    ENDP

```

Dskpatch moet nu een scherm tonen zoals dat van afb. 17-1.

Drive A	Sector 0	
	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	0123456789ABCDEF
00	33 2A 90 49 42 4D 20 20 33 2E 30 00 02 02 01 00	IBM 3.0
10	02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00	00 00 00 00
20	14 00 00 00 00 0F 00 00 00 00 00 00 FA 33 C0 8E	00 00 00 00
30	D0 BC 00 7C BB 78 00 36 C5 37 1E 56 16 53 8E C0	00 00 00 00
40	BF 21 7C B9 0B 00 FC AC 26 80 3D 00 74 03 26 8A	00 00 00 00
50	05 AA E2 F3 8C C0 8E D8 89 47 02 C7 07 21 7C FB	00 00 00 00
60	CD 13 73 03 E9 C7 00 0E 1F A0 10 7C 98 F7 26 16	00 00 00 00
70	7C 03 06 1C 7C 03 06 0E 7C A3 2C 7C A3 2E 7C B8	00 00 00 00
80	20 00 F7 26 11 7C BB 00 02 03 C3 48 F7 F3 01 06	00 00 00 00
90	2E 7C B8 50 00 8E C0 A1 2C 7C E8 A7 00 33 DB B8	00 00 00 00
A0	01 02 E8 B9 00 72 1A 33 FF B9 0B 00 BE D6 7D FC	00 00 00 00
B0	F3 A6 75 0D BF 20 00 BE E1 7D B9 0B 00 F3 A6 74	00 00 00 00
C0	13 BE 77 7D E8 6F 00 32 E4 CD 16 5E 1F 8F 04 8F	00 00 00 00
D0	44 02 CD 19 26 A1 1C 00 BB 00 02 33 D2 F7 F3 FE	00 00 00 00
E0	C0 A2 20 7C A1 2E 7C A3 75 7D B8 70 00 8E C0 33	00 00 00 00
F0	DB A1 2E 7C E8 4D 00 A0 18 7C 2A 06 16 7C FE C0	00 00 00 00

Druk op functietoets, of voer teken of hex-byte in:

Afb. 17-1. Dskpatch met bovenaan de kop.

17.5 Samenvatting

Eindelijk hebben we kennis gemaakt met de ROM BIOS-routines binnenin onze IBM-PC's, en we hebben al twee van deze routines gebruikt om ons te helpen bij het streven naar een volledig Dskpatch-programma.

Eerst hebben we geleerd over INT 10h, functie 6, die we hebben gebruikt om het

scherm te wissen. Ook zagen we (heel kort) dat deze functie meer kan dan wat wij er in dit boek mee doen. U zou haar uiteindelijk bijvoorbeeld kunnen gebruiken om delen van het scherm mee te verschuiven — in Dskpatch of in uw eigen programma's. Daarna hebben we functie 2 van INT 10h gebruikt om de cursor naar de derde regel op het scherm (regel nummer 2) te zetten, waar we onze sector-dump beginnen te schrijven.

Om onze programma's gemakkelijk in het gebruik te maken, hebben we ook verschillende procedures zodanig herschreven dat ze geheugenvariabelen gebruiken inplaats van vaste getallen. Nu kunnen we andere sectoren lezen en op andere manieren de werkwijze van ons programma's veranderen door gewoon een paar centrale getallen in DSKPATCH.ASM te veranderen.

Ten slotte hebben de procedures SCHRIJF__KOP en SCHRIJF__STRING gecreëerd om bovenaan het scherm een kop te kunnen afdrukken. Zoals gezegd, zullen we in het volgende hoofdstuk een verbeterde versie van SCHRIJF__TEK schrijven om de punten in het ASCII-venster van onze afbeelding te vervangen door grafische tekens. En dankzij de modulaire opbouw doen we dat zonder een van de procedures die SCHRIJF__TEK gebruiken te moeten veranderen.

2 6
3 ✓
5 ✓
6 0

18 De definitieve SCHRIJF TEK

- 18.1 Een nieuwe SCHRIJF TEK 204**
- 18.2 Wissen tot het einde van een regel 207**
- 18.3 Samenvatting 209**

We hebben de ROM BIOS-routines in het vorige hoofdstuk goed benut voor het wissen van het scherm en verplaatsen van de cursor. Maar het ROM BIOS biedt nog veel meer mogelijkheden, en sommige daarvan zullen we in dit hoofdstuk zien. Met DOS alleen konden we niet alle 256 tekens afdrukken die de IBM-PC kan laten zien. Daarom geven we in dit hoofdstuk een nieuwe versie van SCHRIJF__TEK die dankzij een andere VIDEO__IO-functie elk teken op het scherm afdruckt. Daarna schrijven we nog een andere nuttige procedure genaamd WIS__TOT__EINDE__REGEL, die de regel vanaf de cursor tot aan de rechterkant van het scherm wist. We zetten die in SCHRIJF__KOP, waar we hem goed kunnen gebruiken om de rest van de regel te wissen. Stel dat we van sector 10 (twee cijfers) naar sector 9 gaan. Er zal dan na het aanroepen van SCHRIJF__KOP een nul van de 10 overblijven terwijl het sectornummer 9 is. WIS__TOT__EINDE__REGEL verwijdert deze nul, en al het andere dat nog op de regel staat.

18.1 Een nieuwe SCHRIJF__TEK

De ROM BIOS-functie 9 van INT 10h schrijft een teken met bijbehorend *attribuut* naar de huidige cursorpositie. Het attribuut bestuurt kenmerken als onderstrepen, knipperen en kleur (zie de beschrijving van de verschillende kleurcodes in uw BASIC-handleiding onder COLOR). Voor Dskpatch zullen we maar twee attributen gebruiken: attribuut 7, het normale attribuut, en attribuut 70h, die een voorgrondkleur nul en een achtergrondkleur 7 geeft, met als gevolg zwarte tekens op een witte achtergrond (inverse video). We kunnen de attributen voor elk teken afzonderlijk instellen, en zullen dat verderop ook doen om een geblokte cursor in inverse video — de *fantomcursor* — te creëren. Voorlopig gebruiken we echter alleen het normale attribuut als we een teken schrijven.

INT 10h, functie 9, schrijft het teken en attribuut naar de huidige cursorpositie. Anders dan DOS zet hij de cursor niet op de plaats van het volgende teken, tenzij hij meer dan één kopie van een teken schrijft. We zullen dit gegeven verderop, in een andere procedure, gebruiken, maar nu willen we maar één kopie van elk teken, dus we zullen zelf de cursor verplaatsen.

Dit is de nieuwe versie van SCHRIJF__TEK, die een teken schrijft en de cursor dan één teken naar rechts zet. Voeg haar in het bestand VIDEO__IO.ASM in.

Listing 18-1. Veranderingen SCHRIJF__TEK in VIDEO__IO.ASM

```

PUBLIC  SCHRIJF_TEK
EXTRN  CURSOR_RECHTS:NEAR

;-----;
; Deze procedure schrijft een teken naar het scherm m.b.v de ROM BIOS- ;
; routines, zodat tekens als backspace worden behandeld als elk ander ;
; teken en worden afgebeeld. ;
; Deze procedure moet eerst wat werk verrichten om de cursorpositie ;
; bij te werken. ;
; ;
;      DL      Byte die op het scherm moet worden afgedrukt ;
; ;
; Gebruikt:      CURSOR_RECHTS ;
;-----;
```

Listing 18-1. *vervolg*

```

SCHRIJF_TEK      PROC      NEAR
    PUSH        AX
    PUSH        BX
    PUSH        CX
    PUSH        DX
    MOV         AH,9          ;uitvoer teken/attribuut aanroepen
    MOV         BH,0          ;instellen op scherpagina 0
    MOV         CX,1          ;schrijf maar een teken
    MOV         AL,DL         ;te schrijven teken
    MOV         BL,7          ;normaal attribuut
    INT         10h           ;schrijf teken en attribuut
    CALL        CURSOR_RECHTS ;nu naar volgende cursorpositie
    POP         DX
    POP         CX
    POP         BX
    POP         AX
    RET
SCHRIJF_TEK      ENDP

```

U zult zich bij het doorlezen van deze procedure misschien hebben afgevraagd wat de instructie MOV BH,0 erin moet. Als u een grafische video-adapter hebt, heeft uw adapter in de normale tekstmodus vier tekstpagina's. We zullen alleen de eerste pagina, pagina 0, gebruiken; vandaar de instructie.

Wat de cursor betreft: SCHRIJF__TEK gebruikt de procedure CURSOR__RECHTS om de cursor een tekenpositie naar rechts te verplaatsen, of naar het begin van de volgende regel als de verplaatsing zich uitstrekt tot voorbij kolom 79. Zet de volgende procedure in CURSOR.ASM:

Listing 18-2. Voeg deze procedure toe aan CURSOR.ASM

```

        PUBLIC  CURSOR_RECHTS
;-----;
; Deze procedure zet de cursor een plaats naar rechts of op de volgende ;
; regel als de cursor aan het einde van een regel stond.                ;
;                                                                           ;
; Gebruikt:          STUUR_CRLF                                           ;
;-----;
CURSOR_RECHTS  PROC      NEAR
    PUSH        AX
    PUSH        BX
    PUSH        CX
    PUSH        DX
    MOV         AH,3          ;lees huidige cursorpositie
    MOV         BH,0          ;op pagina 0
    INT         10h           ;lees cursorpositie
    MOV         AH,2          ;stel nieuwe cursorpositie in
    INC         DL            ;zet kolom op volgende positie
    CMP         DL,79         ;zorg dat kolom <= 79
    JBE         OK
    CALL        STUUR_CRLF    ;ga naar volgende regel
    JMP         KLAAR
OK:          INT         10h
KLAAR:       POP         DX

```


Listing 18-2. *vervolg*

```

        POP      CX
        POP      BX
        POP      AX
        RET
CURSOR_RECHTS  ENDP

```

CURSOR_RECHTS gebruikt twee nieuwe INT 10h-functies. Functie 3 leest de positie van de cursor, en functie 2 verandert de cursorpositie. De procedure gebruikt eerst functie 3 om de cursorpositie te bepalen, die dan in twee bytes wordt gezet, met het kolomnummer in DL en het regelnummer in DH. Daarna hoogt CURSOR_RECHTS het kolomnummer (in DL) op en verplaatst de cursor. Als DL de laatste kolom (79) bevat, verstuurt de procedure een carriage return en line feed om de cursor op de volgende regel te zetten. We hebben deze controle op kolom 79 niet nodig in Dskpatch, maar door hem in CURSOR_RECHTS te zetten, bieden we u een algemene procedure die u in elk van uw eigen programma's kunt gebruiken. Met deze veranderingen moet Dskpatch nu alle 256 tekens afdrukken, zoals afb. 18-1 laat zien.

Drive A	Sector 0	
	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	0123456789ABCDEF
00	EB 2A 90 49 42 4D 20 20 33 2E 30 00 02 02 01 00	6*EIBM 3.0 000
10	02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00	Op 000 0 0
20	14 00 00 00 00 0F 00 00 00 00 00 00 FA 33 C0 8E	91 * .3 Lx
30	D0 BC 00 7C BB 78 00 36 C5 37 1E 56 16 53 8E C0	u i x 6+7 u S A L
40	BF 21 7C B9 0B 00 FC AC 26 80 3D 00 74 03 26 8A	7 ! : d n / & = t & e
50	05 AA E2 F3 8C C0 8E D8 89 47 02 C7 07 21 7C FB	7 r < i L a t e C o l l o ! i J
60	CD 13 73 03 E9 C7 00 0E 1F A0 10 7C 98 F7 26 16	= ! s & o l l f r a b l y z & m
70	7C 03 06 1C 7C 03 06 0E 7C A3 2C 7C A3 2E 7C B8	i v & L i v & f i u i u i j
80	20 00 F7 26 11 7C BB 00 02 03 C3 48 F7 F3 01 06	z & & i j o v H z & o
90	2E 7C B8 50 00 8E C0 A1 2C 7C E8 A7 00 33 DB B8	. i j P A l i i o 3
A0	01 0C E8 B9 00 72 1A 33 FF B9 0B 00 BE D6 7D FC	000 i r > 3 j d f p n
B0	F3 A6 75 0D BF 20 00 BE E1 7D B9 0B 00 F3 A6 74	< u p j d o j d < a t
C0	13 BE 77 7D E8 6F 00 32 E4 CD 16 5E 1F 8F 04 8F	! d w > o 2 z = v A & A
D0	44 02 CD 19 26 A1 1C 00 BB 00 02 33 D2 F7 F3 FE	D o = & i L u o 3 m z & m
E0	C0 A2 20 7C A1 2E 7C A3 75 7D B8 70 00 8E C0 33	L o i i i u u > p A l 3
F0	DB A1 2E 7C E8 4D 00 A0 18 7C 2A 06 16 7C FE C0	i i ! & M a t i * & i i L

C)

Afb. 18-1. Dskpatch met de nieuwe SCHRIJF TEK.

U kunt nagaan of het klopt door een byte te zoeken met een waarde beneden 20h en te kijken of de punt die die waarde eerst in het ASCII-venster genereerde nu is vervangen door een vreemd teken.

Nu gaan we iets doen wat misschien nog interessanter is: een procedure schrijven die een regel vanaf de cursorpositie tot het einde wist.

18.2 Wissen tot het einde van een regel

In het vorige hoofdstuk hebben we de INT 10h-functie 6 in de procedure WIS_SCHERM gebruikt om het scherm schoon te maken. We zeiden toen dat functie 6 kon worden gebruikt om elk rechthoekig venster te wissen. Dat kan zelfs wanneer een venster maar één regel hoog en minder dan één regel lang is, dus we kunnen functie 6 mooi gebruiken om een deel van een regel te verwijderen — tot aan het einde van de regel.

De linkerkant van het venster is, in dit geval, het kolomnummer van de cursor, die we opvragen via functie 3 (ook door CURSOR_RECHTS gebruikt). De rechterkant van het venster is altijd kolom 79. De details kunt u zien in WIS_TOT_EINDE_REGEL; zet de procedure in CURSOR.ASM:

Listing 18-3. Voeg deze procedure toe aan CURSOR.ASM

```

PUBLIC WIS_TOT_EINDE_REGEL
;-----;
; Deze procedure wist de regel vanaf de huidige cursorpositie tot het ;
; einde van die regel. ;
;-----;
WIS_TOT_EINDE_REGEL PROC NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,3           ;lees huidige cursorpositie
    XOR     BH,BH          ;op pagina nul
    INT     10h            ;(X,Y) nu in DL, DH
    MOV     AH,6           ;klaarmaken voor wissen tot einde regel
    XOR     AL,AL          ;maak venster leeg
    MOV     CH,DH          ;allemaal op dezelfde regel
    MOV     CL,DL          ;begin op plaats van de cursor
    MOV     DL,79          ;en stop bij einde van de regel
    MOV     BH,7           ;gebruik normaal attribuut
    INT     10h
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
WIS_TOT_EINDE_REGEL ENDP
```

We zullen deze procedure in SCHRIJF_KOP gebruiken om de rest van de regel te verwijderen wanneer we de andere sectoren lezen (dat komt zo). U kunt WIS_TOT_EINDE_REGEL op geen enkele manier met SCHRIJF_KOP zien werken vóór we de procedures hebben toegevoegd waarmee we een andere sector kunnen lezen en het scherm bijwerken, maar laten we nu eerst SCHRIJF_KOP herschrijven, dan is dat klaar. Breng de volgende wijzigingen in SCHRIJF_KOP in VIDEO_IO.ASM aan, zodat WIS_TOT_EINDE_REGEL aan het einde van de procedure wordt aangeroepen:

Listing 18-4. Veranderingen SCHRIJF_KOP in VIDEO_IO.ASM

```

PUBLIC SCHRIJF_KOP
DATA_SEG SEGMENT PUBLIC
    EXTRN KOPREGEL_NR:BYTE
    EXTRN KOP_DEEL_1:BYTE
    EXTRN KOP_DEEL_2:BYTE
    EXTRN DISKDRIVE_NR:BYTE
    EXTRN HUIDIGE_SECTORNR:WORD
DATA_SEG ENDS
    EXTRN GANAAR_XY:NEAR, WIS_TOT_EINDE_REGEL:NEAR
;-----;
; Deze procedure schrijft de kop met diskdrive en sector-nummer. ;
; ;
; Gebruikt: GANAAR_XY, SCHRIJF_STRING, SCHRIJF_TEK, SCHRIJF_DECIMAAL;
; WIS_TOT_EINDE_REGEL ;
; Leest: KOPREGEL_NR, KOP_DEEL_1, KOP_DEEL_2 ;
; DISKDRIVE_NR, HUIDIGE_SECTORNR ;
;-----;
SCHRIJF_KOP PROC NEAR
    PUSH DX
    XOR DL,DL ;zet cursor op regelnummer kop
    MOV DH,KOPREGEL_NR
    CALL GANAAR_XY
    LEA DX,KOP_DEEL_1
    CALL SCHRIJF_STRING
    MOV DL,DISKDRIVE_NR
    ADD DL,'A' ;druk drives A, B, ... af
    CALL SCHRIJF_TEK
    LEA DX,KOP_DEEL_2
    CALL SCHRIJF_STRING
    MOV DX,HUIDIGE_SECTORNR
    CALL SCHRIJF_DECIMAAL
    CALL WIS_TOT_EINDE_REGEL ;wis rest van sectornummer
    POP DX RET
SCHRIJF_KOP ENDP

```

Deze wijziging brengt meteen de definitieve versie van SCHRIJF_KOP tot stand en betekent dat het bestand CURSOR.ASM klaar is. Aan Dskpatch ontbreken echter nog allerlei belangrijke onderdelen. In het volgende hoofdstuk gaan we verder en voegen er de centrale verdeler voor toetsopdrachten bij, zodat we F1 en F2 kunnen indrukken wanneer we andere sectoren van de schijf willen lezen.

18.3 Samenvatting

Dit hoofdstuk was vrij gemakkelijk, met weinig nieuwe informatie of trucs. Wel hebben we geleerd hoe de INT 20h-functie 9 in het ROM BIOS moet worden gebruikt om elk willekeurig teken op het scherm te kunnen afdrukken.

Daarbij zagen we ook hoe de cursorpositie met INT 20h, functie 3, kan worden gelezen en hoe we de cursor na het schrijven van een teken een plaats naar rechts kunnen zetten. De reden daarvoor: functie 9 van INT 20h zet de cursor niet verder na een teken te hebben geschreven, tenzij ze meer dan één kopie van het teken afdrukt. Ten slotte hebben we functie 6 van INT 20h aan het werk gezet om een deel van slechts één regel te wissen.

In het volgende hoofdstuk moeten we er weer flink tegenaan om de centrale verdeel-procedure op te zetten.

19 De verdeelprocedure

19.1 De Verdeler 211

19.2 Andere sectoren lezen 217

19.3 Opzet van de volgende hoofdstukken 219

In elke taal is het prettig om een goed geschreven programma te hebben dat iets doet, maar om een programma echt tot leven te brengen moeten we het interactief maken. Het is de mens eigen om te zeggen: 'Als jij dit doet, doe ik dat', dus we zullen dit hoofdstuk gebruiken om Dskpatch tot op zekere hoogte interactief te maken. We zullen een eenvoudige toetsenbord-invoerprocedure en een centrale verdeler schrijven. Deze laatstgenoemde procedure zal tot taak hebben om bij elke ingedrukte toets de juiste procedure aan te roepen. Als we bijvoorbeeld op de F1-toets drukken om de vorige sector te lezen en af te beelden, zal de verdeelroutine een procedure genaamd `VORIGE_SECTOR` aanroepen. Daartoe zullen we Dskpatch op verschillende punten veranderen. We beginnen met het aanmaken van `VERDELER`, de centrale verdeelprocedure, en nog wat procedures voor het indelen van de afbeelding. Daarna voegen we twee nieuwe procedures toe, genaamd `VORIGE_SECTOR` en `VOLGENDE_SECTOR`, die we via `VERDELER` zullen aanroepen.

19.1 De Verdeler

De Verdeler wordt het centrale regelorgaan voor Dskpatch, dat alle invoer en wijzigingen via het toetsenbord regelt. `VERDELER` zal tot taak hebben om tekens te lezen en dan het werk te verdelen onder andere procedures. U ziet zo hoe `VERDELER` zijn werk doet, maar eerst kijken we hoe hij in Dskpatch past.

`VERDELER` krijgt een eigen promptregel, vlak onder de halve sector-afbeelding, waar de cursor wacht op invoer van het toetsenbord. In onze eerste versie van de toetsenbordinvoer-procedure zult u geen hex-getallen kunnen invoeren; dat komt nog. Dit zijn de eerste wijzigingen in `DSKPATCH.ASM`; ze voegen de gegevens voor een promptregel toe:

Listing 19-1. Toevoegingen aan `DATA_SEG` in `DSKPATCH.ASM`

```
KOPREGEL_NR      DB      0
KOP_DEEL_1       DB      'Drive ',0
KOP_DEEL_2       DB      '          Sector ',0
PUBLIC PROMPTREGEL_NR, EDITOR_PROMPT
PROMPTREGEL_NR   DB      21
EDITOR_PROMPT    DB      'Druk op functietoets, of voer'
                 DB      ' teken of hex-byte in: ',0
```

Verderop voegen we nog meer prompts toe voor dingen als het invoeren van een nieuw sectornummer, dus we maken het ons gemakkelijker door een gemeenschappelijke procedure te schrijven met de naam `SCHRIJF_PROMPTREGEL` die elke promptregel afdruckt. Elke procedure die `SCHRIJF_PROMPTREGEL` gebruikt, zal hem voorzien van het adres van de prompt (hier het adres van `EDITOR_PROMPT`) en dan de prompt schrijven op regel 21 (omdat `PROMPTREGEL_NR` 21 is). Deze nieuwe versie van `DISK_PATCH` (in `DSKPATCH.ASM`) gebruikt `SCHRIJF_PROMPTREGEL` bijvoorbeeld vlak voor hij `VERDELER` aanroept.

Listing 19-2. Toevoegingen aan `DISK_PATCH` in `DSKPATCH.ASM`

```
EXTRN WIS_SCHERM:NEAR, LEES_SECTOR:NEAR
EXTRN START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
EXTRN SCHRIJF_PROMPTREGEL:NEAR, VERDELER:NEAR
```

Listing 19-2. *vervolg*

```
DISK_PATCH      PROC    NEAR
                CALL    WIS_SCHERM
                CALL    SCHRIJF_KOP
                CALL    LEES_SECTOR
                CALL    START_SEC_AFB
                LEA      DX,EDITOR_PROMPT
                CALL    SCHRIJF_PROMPTREGEL
                CALL    VERDELER
                INT      20h
DISK_PATCH      ENDP
```

De Verdeler zelf is een vrij eenvoudig programma, maar we hebben er wel wat trucs in toegepast. Hieronder staat onze eerste versie van het bestand VERDEEL.ASM.

Listing 19-3. Het nieuwe bestand VERDEEL.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC

        PUBLIC VERDELER
        EXTRN  LEES_BYTE:NEAR

;-----;
; Dit is de centrale verdeler. Bij normaal wijzigen (editen) en ;
; bekijken, leest deze procedure tekens van het toetsenbord en, als het ;
; teken een opdrachttoets (b.v. een cursortoets) is, roept VERDELER ;
; procedures aan die het eigenlijke werk doen. Dit gebeurt bij speciale ;
; toetsen die vermeld staan in de tabel VERDEEL_TABEL, waarin de ;
; adressen van de procedures vlak na de toetsnamen zijn opgeslagen. ;
; Als het teken geen speciale toets is, moet het rechtstreeks in de ;
; sectorbuffer worden gezet -- dit is de edit-modus. ;
; ;
; Gebruikt:      LEES_BYTE ;
;-----;

VERDELER      PROC    NEAR
                PUSH    AX
                PUSH    BX
VERDEEL_LUS:
                CALL    LEES_BYTE      ;lees teken naar AX
                OR      AH,AH          ;AX = 0 als geen teken gelezen, -1
                JZ      VERDEEL_LUS    ;voor uitgebreide code
                JS      SPECIALE_TOETS ;geen teken gelezen, probeer opnieuw
                JZ      VERDEEL_LUS    ;lees uitgebreide code
;doe voorlopig niets met het teken
                JMP     VERDEEL_LUS    ;lees nog een teken

SPECIALE_TOETS:
                CMP     AL,68          ;F10--stop?
                JE      EINDE_VERDEEL ;ja, einde

                LEA     BX,VERDEEL_TABEL ;gebruik BX om tabel door te kijken
```


Listing 19-3. *vervolg*

```

SPECIALE_LUS:
    CMP     BYTE PTR [BX],0           ;einde van tabel?
    JE      NIET_IN_TABEL             ;ja, toets stond niet in tabel
    CMP     AL,[BX]                   ;is het deze tabel-ingang?
    JE      VERDEEL                   ;ja, dan werk verdelen
    ADD     BX,3                       ;nee, probeer volgende ingang
    JMP     SPECIALE_LUS              ;controleer volgende tabel-ingang

VERDEEL:
    INC     BX                        ;wijs naar adres van procedure
    CALL    WORD PTR [BX]             ;roep procedure aan
    JMP     VERDEEL_LUS               ;wacht op een andere toets

NIET_IN_TABEL:
    JMP     VERDEEL_LUS               ;doe niets, lees gewoon volgende teken

EINDE_VERDEEL:
    POP     BX
    POP     AX
    RET

VERDELER    ENDP

CODE_SEG    ENDS

DATA_SEG     SEGMENT PUBLIC

CODE_SEG     SEGMENT PUBLIC
    EXTRN    VOLGENDE_SECTOR:NEAR      ;in DISK_IO.ASM
    EXTRN    VORIGE_SECTOR:NEAR        ;in DISK_IO.ASM
CODE_SEG     ENDS

;-----;
; Deze tabel bevat de toegestane uitgebreide ASCII-toetsen en de ;
; adressen van de procedures die moeten worden aangeroepen wanneer een ;
; toets wordt ingedrukt. ;
; De tabel heeft de volgende indeling: ;
; DB 72 ;uitgebreide code voor cursor omhoog ;
; DW OFFSET CGROEP:FANTOOM_OMHOOG ;
;-----;
VERDEEL_TABEL LABEL BYTE
    DB 59 ;F1
    DW OFFSET CGROEP:VORIGE_SECTOR
    DB 60 ;F2
    DW OFFSET CGROEP:VOLGENDE_SECTOR
    DB 0 ;einde van de tabel
DATA_SEG     ENDS

END

```

VERDEEL_TABEL bevat de uitgebreide ASCII-codes voor de F1- en de F2-toets. Elke code wordt gevolgd door het adres van de procedure die VERDELER aanroept wanneer hij die bepaalde code leest. Wanneer bijvoorbeeld LEES_BYTE, die door VERDELER wordt aangeroepen, een F1-toets leest (uitgebreide code 59), roept

VERDELER de procedure VORIGE_SECTOR aan.

De adressen van de procedures die we VERDELER willen laten aanroepen, staan in de verdeeltabel, dus we hebben een pseudo-op, OFFSET, gebruikt om ze binnen te halen. De regel:

```
DW      OFFSET CGROEP:VORIGE_SECTOR
```

vertelt de assembler bijvoorbeeld dat hij de *offset* van onze procedure VORIGE_SECTOR moet gebruiken. Deze offset wordt gerekend vanaf het begin van onze groep CGROEP; vandaar dat we de CGROEP: voor de procedurenaam moeten hebben. Als we CGROEP daar niet hadden gezet, zou de assembler het adres van VORIGE_SECTOR berekenen vanaf het begin van het codesegment, en dat willen we misschien niet. (Zoals hier blijkt, is deze CGROEP niet per se noodzakelijk omdat het codesegment het eerst in ons programma wordt geladen. Maar voor alle duidelijkheid zetten we OFFSET CGROEP: er toch maar bij.)

Merk op dat VERDEEL_TABEL zowel byte- als woordgegevens bevat. Dit leidt tot wat overwegingen. Tot dusver hebben we ons steeds beziggehouden met tabellen van een bepaald type: of allemaal woorden, of allemaal bytes. Maar hier hebben we allebei, dus we moeten de assembler vertellen welk soort gegevens hij moet verwachten als we een CMP- of CALL-instructie gebruiken. In het geval van een instructie als

```
CMP      [BX],0
```

weet de assembler niet of we woorden of bytes willen vergelijken. Maar als we de instructie zo schrijven:

```
CMP      BYTE PTR [BX],0
```

vertellen we de assembler dat BX naar een byte wijst, en dat we een vergelijking van bytes willen. Op dezelfde manier zou de instructie CMP WORD PTR [BX],0 woorden vergelijken. Anderzijds zou een instructie als CMP AL,[BX] geen probleem vormen omdat AL een byte-register is, en de assembler zonder dat hem dat wordt verteld ook wel weet dat we bytes willen vergelijken.

Denk er ook aan dat een CALL-instructie zowel NEAR als FAR kan zijn. NEAR CALL heeft één woord voor het adres nodig, terwijl FAR CALL er twee nodig heeft. De instructie:

```
CALL      WORD PTR [BX]
```

vertelt de assembler hier, met WORD PTR, dat [BX] naar een woord wijst en dus een korte CALL moet genereren en het woord gebruiken dat [BX] als het adres aanwijst — een adres dat we in VERDEEL_TABEL hebben opgeslagen. (Voor een FAR CALL, dat een adres van twee woorden gebruikt, zouden we de instructie CALL DWORD PTR [BX] gebruiken. DWORD betekent *Double Word*, twee woorden.) Zoals u in hoofdstuk 22 zult zien, kunnen we gemakkelijk meer toetsopdrachten aan Dskpatch toevoegen door er gewoon meer procedures in te zetten en nieuwe ingangen in VERDEEL_TABEL te zetten. Voorlopig moeten we echter nog vier nieuwe procedures toevoegen vóór we deze nieuwe versie van Dskpatch kunnen testen.

LEES_BYTE, SCHRIJF_PROMPTREGEL, VORIGE_SECTOR en VOLGENDE_SECTOR ontbreken nog.

LEES_BYTE is een procedure die tekens en uitgebreide ASCII-tekens van het toetsenbord leest. De uiteindelijke versie zal speciale toetsen (zoals de functie- en cursor-toetsen), ASCII-tekens en hex-getallen van twee cijfers kunnen lezen. Nu maken we eerst een eenvoudige versie van LEES_BYTE die een teken of een speciale toets leest. Dit is de eerste versie van TBD_IO.ASM, het bestand waarin we al onze procedures voor het lezen van het toetsenbord zullen zetten.

Listing 19-4. Het nieuwe bestand TBD_IO.ASM

```
CGROEP  GROUP  CODE_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
        PUBLIC  LEES_BYTE
;-----;
; Deze procedure leest een enkel ASCII-teken. Dit is nog maar een ;
; testversie van LEES_BYTE. ;
; ;
; Zet byte in          AL      Code van teken (tenzij AH = 0) ;
;                      AH      1 als ASCII-teken gelezen ;
;                      -1 als speciale toets gelezen ;
;-----;
LEES_BYTE      PROC      NEAR
        MOV      AH,7          ;lees teken zonder echo
        INT      21h          ;en zet het in AL
        OR       AL,AL        ;is 't een uitgebreide code?
        JZ       UITGEBREIDE_CODE ;ja
NIET_UTGEBREID:
        MOV      AH,1          ;geef normaal ASCII-teken aan
KLAAR_MET_LEZEN:
        RET
UITGEBREIDE_CODE:
        INT      21h          ;lees de uitgebreide code
        MOV      AH,OFFh      ;geef uitgebreide code aan
        JMP      KLAAR_MET_LEZEN
LEES_BYTE      ENDP
CODE_SEG      ENDS

        END
```

We zetten SCHRIJF_PROMPTREGEL als volgt in VIDEO_IO.ASM:

Listing 19-5. Voeg deze procedure toe aan VIDEO_IO.ASM

```
        PUBLIC  SCHRIJF_PROMPTREGEL
        EXTRN   WIS_TOT_EINDE_REGEL:NEAR
        EXTRN   GANAAR_XY:NEAR
DATA_SEG  SEGMENT PUBLIC
        EXTRN   PROMPTREGEL_NR:BYTE
DATA_SEG  ENDS
```

Listing 19-5. *vervolg*

```

;-----;
; Deze procedure schrijft de promptregel naar het scherm en wist tot ;
; het einde van de regel ;
; ;
; DS:DX Adres van de promptregel-melding ;
; ;
; Gebruikt: SCHRIJF_STRING, WIS_TOT_EINDE_REGEL, GANAAR_XY ;
; Leest: PROMPTREGEL_NR ;
;-----;
SCHRIJF_PROMPTREGEL PROC NEAR
    PUSH    DX
    XOR     DL,DL
    MOV     DH,PROMPTREGEL_NR ;schrijf de promptregel en
                                ;zet daar de cursor heen
    CALL    GANAAR_XY
    POP     DX
    CALL    SCHRIJF_STRING
    CALL    WIS_TOT_EINDE_REGEL
    RET
SCHRIJF_PROMPTREGEL ENDP

```

Deze procedure stelt eigenlijk niet zo veel voor. Hij zet de cursor aan het begin van de promptregel, die we (in DSKPATCH.ASM) op regel 21 hebben gezet. Daarna schrijft hij de promptregel en wist tot het einde van de regel. De cursor staat aan het einde van de prompt wanneer SCHRIJF_PROMPTREGEL klaar is, en de rest van de regel wordt gewist door WIS_TOT_EINDE_REGEL.

19.2 Andere sectoren lezen

Ten slotte moeten we nog de procedures VORIGE_SECTOR en VOLGENDE_SECTOR hebben om de vorige en volgende sectoren van de schijf te lezen en af te beelden. Zet deze twee procedures in DISK_IO.ASM:

Listing 19-6. Voeg deze procedure toe aan DISK_IO.ASM

```

PUBLIC VORIGE_SECTOR
EXTRN  START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
EXTRN  SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN  HUIDIGE_SECTORNR:WORD, EDITOR_PROMPT:BYTE
DATA_SEG ENDS
;-----;
; Deze procedure leest, zo mogelijk, de vorige sector. ;
; ;
; Gebruikt: SCHRIJF_KOP, LEES_SECTOR, START_SEC_AFB ;
;           SCHRIJF_PROMPTREGEL ;
; Leest:    HUIDIGE_SECTORNR, EDITOR_PROMPT ;
; Schrijft: HUIDIGE_SECTORNR ;
;-----;
VORIGE_SECTOR PROC NEAR
    PUSH    AX
    PUSH    DX
    MOV     AX,HUIDIGE_SECTORNR ;vraag huidige sectornummer op

```


Listing 19-6. *vervolg*

```

OR      AX,AX                      ;niet verlagen als reeds 0
JZ      SECTOR_NIET_VERLAGEN
DEC     AX
MOV     HUIDIGE_SECTORNR,AX        ;bewaar nieuwe sectornummer
CALL    SCHRIJF_KOP
CALL    LEES_SECTOR
CALL    START_SEC_AFB              ;beeld nieuwe sector af
LEA     DX,EDITOR_PROMPT
CALL    SCHRIJF_PROMPTREGEL
SECTOR_NIET_VERLAGEN:
POP     DX
POP     AX
RET
VORIGE_SECTOR      ENDP
PUBLIC VOLGENDE_SECTOR
EXTRN  START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
EXTRN  SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG      SEGMENT PUBLIC
EXTRN  HUIDIGE_SECTORNR:WORD, EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
;-----;
; Leest de volgende sector. ;
; ;
; Gebruikt:  SCHRIJF_KOP, LEES_SECTOR, START_SEC_AFB ;
;           SCHRIJF_PROMPTREGEL ;
; Leest:    HUIDIGE_SECTORNR, EDITOR_PROMPT ;
; Schrijft: HUIDIGE_SECTORNR ;
;-----;
VOLGENDE_SECTOR PROC    NEAR
PUSH    AX
PUSH    DX
MOV     AX,HUIDIGE_SECTORNR
INC     AX                      ;ga naar volgende sector
MOV     HUIDIGE_SECTORNR,AX
CALL    SCHRIJF_KOP
CALL    LEES_SECTOR
CALL    START_SEC_AFB          ;beeld nieuwe sector af
LEA     DX,EDITOR_PROMPT
CALL    SCHRIJF_PROMPTREGEL
POP     DX
POP     AX
RET
VOLGENDE_SECTOR ENDP

```

Nu bent u klaar om alle bestanden die we hebben aangemaakt en veranderd te assembleren: Dskpatch, Video__io, Tbd__io, Dispatch en Disk__io. Denk er bij het linken van de Dskpatch-bestanden aan dat het er nu zeven zijn: Dskpatch, Toon__sec, Disk__io, Video__io, Tbd__io, Verdeel en Cursor.

Voor wie Make gebruikt zijn hier de toevoegingen die nodig zijn voor het bestand Dskpatch (de backslash achteraan de vierde regel van onderen betekent voor Make dat we op de volgende regel doorgaan met de reeks bestanden):

Listing 19-7. Het gewijzigde Make-bestand DSKPATCH

```
dskpatch.obj:  dskpatch.asm
               masm dskpatch;

cursor.obj:    cursor.asm
               masm cursor;

verdeel.obj:   verdeel.asm
               masm verdeel;

tbd_io.obj:    tbd_io.asm
               masm tbd_io;

disk_io.obj:   disk_io.asm
               masm disk_io;
toon_sec.obj:  toon_sec.asm
               masm toon_sec;

video_io.obj:  video_io.asm
               masm video;

dskpatch.com:  dskpatch.obj disk_io.obj toon_sec.obj video_io.obj \
               cursor.obj verdeel.obj tbd_io.obj
               link dskpatch disk_io toon_sec video_io cursor verdeel tbd_io;
               exe2bin dskpatch dskpatch.com
```

Beschikt u niet over Make, dan kunt het volgende korte batch-bestand schrijven voor het linken en aanmaken van uw .COM-bestand:

```
LINK DSKPATCH DISK_IO TOON_SEC VIDEO_IO CURSOR VERDEEL TBD_IO;
EXE2BIN DSKPATCH DSKPATCH.COM
```

Als we er meer bestanden aan toevoegen, hoeft u alleen dit batchbestand te wijzigen, in plaats van iedere keer die lange reeks te linken bestanden te tikken wanneer u het .COM-programma wilt wijzigen.

Deze versie van Dskpatch heeft drie actieve toetsen: F1 leest en toont de vorige sector, en stopt bij sector 0; F2 leest de volgende sector; F10 beëindigt Dskpatch. Probeer die toetsen eens. Uw scherm moet er nu ongeveer uitzien zoals in afb. 19-1.

19.3 Opzet van de volgende hoofdstukken

We hebben in dit hoofdstuk veel meer terrein bestreken dan gewoonlijk, en in dit opzicht hebt u een voorproefje gehad van de opzet die we in de hoofdstukken 20 tot en met 29 aanhouden. Van nu af zult u vrij snel voortgang maken, zodat we u meer voorbeelden kunnen laten zien van hoe u grote programma's moet schrijven. U zult ook meer procedures zien, die u in uw eigen programma's kunt gebruiken.

Deze hoofdstukken zijn er voor u om er iets van te leren, vandaar de vele procedures die erin staan. Maar in de twee laatste hoofdstukken van het boek komen er weer nieuwe onderwerpen om te leren, dus ga door of sla (als u wilt) de overige hoofdstukken over Dskpatch over tot u zover bent dat u uw eigen programma's kunt schrijven.

Drive A

Sector 0

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	2A	90	49	42	4D	20	20	33	2E	30	00	02	02	01	00	IBM 3.0
10	02	70	00	D0	02	FD	02	00	09	00	02	00	00	00	00	00	0 0
20	14	00	00	00	00	0F	00	00	00	00	00	00	FA	33	C0	8E	3 4
30	D0	BC	00	7C	BB	78	00	36	C5	37	1E	56	16	53	8E	C0	6+7
40	BF	21	7C	B9	0B	00	FC	AC	26	80	3D	00	74	03	26	8A	SA L
50	05	AA	E2	F3	8C	C0	8E	D8	89	47	02	C7	07	21	7C	FB	n/g= t&e
60	CD	13	73	03	E9	C7	00	0E	1F	A0	10	7C	98	F7	26	16	LA te G
70	7C	03	06	1C	7C	03	06	0E	7C	A3	2C	7C	A3	2E	7C	B8	ly&=
80	20	00	F7	26	11	7C	BB	00	02	03	C3	48	F7	F3	01	06	u, i, i
90	2E	7C	B8	50	00	8E	C0	A1	2C	7C	E8	A7	00	33	DB	B8	H&=
A0	01	02	E8	B9	00	72	1A	33	FF	B9	0B	00	BE	D6	7D	FC	3
B0	F3	A6	75	0D	BF	20	00	BE	E1	7D	B9	0B	00	F3	A6	74	r+3
C0	13	BE	77	7D	E8	6F	00	32	E4	CD	16	5E	1F	8F	04	8F	d
D0	44	02	CD	19	26	A1	1C	00	BB	00	02	33	D2	F7	F3	FE	2Σ=^A
E0	C0	A2	20	7C	A1	2E	7C	A3	75	7D	B8	70	00	8E	C0	33	il q
F0	DB	A1	2E	7C	E8	4D	00	A0	18	7C	2A	06	16	7C	FE	C0	3 p A L

Druk op functietoets, of voer teken of hex-byte in:

Afb. 19-1. Dskpatch met de promptregel.

Als u klaar bent om terug te keren, zult u allerlei nuttige informatie over programmeren aantreffen.

Als u ernaar snakt om eigen procedures te gaan schrijven, moet u natuurlijk het volgende hoofdstuk lezen. U ziet daar een aantal tips, en we bieden u de gelegenheid de procedures in de volgende hoofdstukken te schrijven door u voldoende details te geven om vooruit te komen.

Vanaf hoofdstuk 21 tonen we allerlei verschillende procedures en laten we u zelf uitzoeken hoe ze werken. Waarom? Daar zijn twee redenen voor, beide bedoeld om u op eigen benen te laten staan en op weg te helpen naar het programmeren in assembleertaal. Ten eerste dient u een bibliotheek met procedures te hebben die u in uw eigen programma's kunt gebruiken; daarvoor moet u over u over de nodige vaardigheid beschikken. Ten tweede willen we u met dit grote programmavoorbeeld niet alleen laten zien hoe u een groot programma moet schrijven, maar u er ook vertrouwd mee maken.

Werk de rest van het boek dus maar door zoals het u goeddunkt. Hoofdstuk 20 is voor degenen die graag zelf programma's willen gaan schrijven. In hoofdstuk 21 keren we terug naar Dskpatch en bouwen de procedures op voor het schrijven en verplaatsen van wat we een fantoomcursor noemen: een cursor in inverse video voor de hex- en ASCII-afbeeldingen.

20 Een programmeer- uitdaging

20.1 De fantoomcursors 222

20.2 Eenvoudige wijzigingen 223

**20.3 Andere toevoegingen en wijzigingen in
Dskpatch 224**

Dit boek bevat nog zes hoofdstukken met procedures. Als u zelf de weg wilt proberen te vinden, lees dan dit hoofdstuk. We zullen hier een koers voor u uitzetten en aangeven hoe u door de hoofdstukken 21 en 22 heen moet. Daarna kunt u proberen de procedures in elk hoofdstuk te schrijven voor u ze doorleest. Als u nu nog niet wilt proberen om delen van Dskpatch te schrijven, sla dit hoofdstuk dan over. Het is erg kort en laat veel aan uw verbeelding over.

Als u besluit dit hoofdstuk door te lezen, kunt u dat het beste zo doen: Lees een paragraaf en probeer dan zelf de overeenkomstige veranderingen in Dskpatch aan te brengen. Als u het gevoel hebt voldoende vooruitgang te hebben geboekt, lees dan het hoofdstuk met dezelfde titel als van de paragraaf door. Nadat u het overeenkomstige hoofdstuk hebt gelezen, kunt u doorgaan met de volgende paragraaf.

N.B. U kunt het beste nu een kopie van al uw bestanden maken voor u veranderingen gaat aanbrengen. Als u dan bij hoofdstuk 21 bent, kunt u kiezen of u met de veranderingen meegaat of uw eigen versie gebruikt.

20.1 De fantoomcursors

In hoofdstuk 21 zetten we twee fantoomcursors op het scherm: één in het hex-venster en één in het ASCII-venster. Een fantoomcursor is net zo iets als een gewone cursor, maar knippert niet en de achtergrond wordt wit en de tekens zwart, zoals u in afb. 20-1 ziet.

Drive A	Sector 0																0123456789ABCDEF													
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F														
00	02	70	00	D0	02	FD	02	00	09	00	02	00	00	00	00	00	IBM 3.0													
10	14	00	00	00	00	0F	00	00	00	00	00	00	FA	33	C0	8E	Op 000000													
20	D0	BC	00	7C	BB	78	00	36	C5	37	1E	56	16	53	8E	C0	X 6740													
30	BF	21	7C	B9	0B	00	FC	AC	26	80	3D	00	74	03	26	8A	L 6740													
40	05	AA	E2	F3	8C	C0	8E	D8	89	47	02	C7	07	21	7C	FB	L 6740													
50	CD	13	73	03	E9	C7	00	0E	1F	A0	10	7C	98	F7	26	16	L 6740													
60	7C	03	06	1C	7C	03	06	0E	7C	A3	2C	7C	A3	2E	7C	B8	L 6740													
70	20	00	F7	26	11	7C	BB	00	02	03	C3	48	F7	F3	01	06	L 6740													
80	2E	7C	B8	50	00	8E	C0	A1	2C	7C	E8	A7	00	33	DB	B8	L 6740													
90	01	02	E8	B9	00	72	1A	33	FF	B9	0B	00	BE	D6	7D	FC	L 6740													
A0	F3	A6	75	0D	BF	20	00	BE	E1	7D	B9	0B	00	F3	A6	74	L 6740													
B0	13	BE	77	7D	E8	6F	00	32	E4	CD	16	5E	1F	8F	04	8F	L 6740													
C0	44	02	CD	19	26	A1	1C	00	BB	00	02	33	D2	F7	F3	FE	L 6740													
D0	C0	A2	20	7C	A1	2E	7C	A3	75	7D	B8	70	00	8E	C0	33	L 6740													
E0	DB	A1	2E	7C	E8	4D	00	A0	18	7C	2A	06	16	7C	FE	C0	L 6740													
F0																	L 6740													

Druk op functietoets, of voer teken of hex-byte in:

Afb. 20-1. Een afbeelding met fantoomcursors.

De fantoomcursor in het hex-venster is vier tekens breed, die in het ASCII-venster maar één.

Hoe maken we een fantoomcursor? Elk teken op het scherm heeft een *attribuut*-byte. Deze byte vertelt uw IBM-PC hoe elk teken moet worden afgebeeld. Een attribuutcode 7h laat een normaal teken zien, terwijl 70h een teken in inverse video laat zien. Het laatste is precies wat we willen voor de fantoomcursor, dus de vraag is: hoe veranderen we het attribuut van onze tekens in 70h?

INT 10h, functie 9, schrijft zowel een teken als een attribuut naar het scherm, en INT 10h, functie 8, leest de code van het teken op de huidige cursorpositie. We kunnen een fantoomcursor in het hex-venster maken door de volgende stappen te zetten:

- Bewaar de plaats van de echte cursor (gebruik INT 10H, functie 3, voor het lezen van de cursorpositie en opslaan daarvan in variabelen).
- Zet de echte cursor aan het begin van de fantoomcursor in het hex-venster.
- Lees voor de volgende vier tekens de tekencode (functie 8) en schrijf zowel het teken als zijn attribuut (zet attribuut op 70h).
- Zet ten slotte de oude cursor terug op zijn plaats.

De fantoomcursor in het ASCII-venster schrijven we op vrijwel dezelfde manier. Hebt u eenmaal een werkende fantoomcursor in het hex-venster, dan kunt u de extra code voor het ASCII-venster er bijzetten. Vergeet niet dat uw eerste poging maar tijdelijk is. Hebt u eenmaal een programma met een fantoomcursor die werkt, dan kunt u terug en uw veranderingen zo herschrijven dat u een klein aantal procedures hebt die het werk doen. Bekijk de procedures in hoofdstuk 21 als u klaar bent, om te zien hoe het ook kan.

20.2 Eenvoudige wijzigingen

Als we eenmaal onze fantoomcursors hebben, willen we ze over het scherm kunnen verplaatsen. We moeten hier goed op de grenscondities letten om de fantoomcursors binnen elk van de twee vensters te houden. We willen ook dat onze twee fantoomcursors tegelijkertijd worden verplaatst omdat ze de hex- en de ASCII-voorstelling van hetzelfde ding betreffen.

Hoe kunnen we de fantoomcursor verplaatsen? Elk van de vier cursortoetsen op het toetsenbord verstuurt een speciaal functienummer: 72 voor cursor omhoog, 80 voor cursor omlaag, 75 voor cursor naar links en 77 voor cursor naar rechts. Dit zijn de nummers die we aan VERDEEL__TABEL moeten toevoegen, samen met de adressen van de vier procedures om de fantoomcursors in deze vier richtingen te kunnen verplaatsen.

Om elke fantoomcursor echt te verplaatsen, moet u ze verwijderen, de twee coördinaten ervan veranderen en ze weer op het scherm zetten. Als u het schrijven van de fantoomcursors goed hebt opgezet, zullen de vier procedures voor het verplaatsen ervan niet zo moeilijk zijn.

Iedere keer als u een teken op het toetsenbord intikt, moet Dskpatch dit teken lezen en de byte onder de fantoomcursor vervangen door het zoëven gelezen teken. Dit moet u doen om eenvoudige wijzigingen te kunnen aanbrengen:

- Lees een teken van het toetsenbord.
- Verander het hex-getal in het hex-venster en het teken in het ASCII-venster zodat

die overeenkomen met het zojuist gelezen teken.

- Verander de byte in de sectorbuffer SECTOR.

Een eenvoudige tip: u hoeft niet veel te veranderen om wijzigingen op het scherm te kunnen aanbrengen (editen). Voor Verdeel is daarvoor weinig meer nodig dan het aanroepen van een nieuwe procedure (we hebben hem EDIT__BYTE genoemd) die het meeste werk doet. EDIT__BYTE is verantwoordelijk voor veranderingen op het scherm zowel als in SECTOR.

20.3 Andere toevoegingen en wijzigingen in Dskpatch

Van hoofdstuk 23 tot en met 27 worden de wijzigingen wat lastiger en ingewikkelder. Als u nog steeds van plan bent uw eigen versie te schrijven, bedenk dan wat u Dskpatch nog meer wilt zien doen dan hij nu al doet. Wij zijn voor de komende hoofdstukken van de volgende gedachte uitgegaan.

We willen een nieuwe versie van LEES__BYTE, die óf een teken óf een hex-getal van twee cijfers leest en wacht tot we de Enter-toets hebben ingedrukt voor hij een teken aan Verdeel doorgeeft. Dit onderdeel van ons 'verlanglijstje' is niet zo eenvoudig als het klinkt, en we zullen twee hoofdstukken (23 en 24) aan dit probleem besteden. In hoofdstuk 25 gaan we fouten opsporen, en in hoofdstuk 26 leren we daarna hoe gewijzigde sectoren naar de schijf kunnen worden teruggeschreven met de DOS-functie INT 26h, die analoog is aan INT 25h die we gebruikten om een sector van de schijf te lezen. (We zullen in hoofdstuk 26 niet op leesfouten controleren, maar u vindt die foutcontroles wel in de schijf-versie van Dskpatch die bij dit boek te verkrijgen is.)

Ten slotte brengen we, in hoofdstuk 27, enige veranderingen aan in Dskpatch om de andere helft van onze sector-afbeelding te kunnen zien. Deze veranderingen hebben niet tot gevolg dat we de cursor zo vrijelijk door de afbeelding kunnen verplaatsen als we zouden willen, maar ook hier geldt weer dat de veranderingen wel in de schijf-versie van Dskpatch staan.

21 De fantoomcursors

- 21.1 De fantoomcursors 226**
- 21.2 Tekenattributen veranderen 231**
- 21.3 Samenvatting 232**

In dit hoofdstuk zetten we de procedures op voor het schrijven en verwijderen van een fantoomcursor in het hex-venster en een in het ASCII-venster. Een fantoomcursor wordt zo genoemd omdat het niet de cursor is van de hardware van de PC; het is een schaduw. . . zij het een nogal ongebruikelijke omdat hij het teken in inverse video weergeeft en de achtergrond wit en het teken zwart maakt. In het hex-venster hebben we ruimte om deze cursor vier tekens breed te maken zodat hij gemakkelijk leesbaar is. In het ASCII-venster wordt de fantoomcursor maar één teken breed omdat er geen ruimte tussen twee tekens is.

We hebben hier nogal wat procedures te bespreken, dus we zullen het allemaal beknopt doen.

21.1 De fantoomcursors

START_SEC_AFB is de enige procedure die we hebben die de sector-afbeelding verandert. Bij opstarten van Dskpatch verschijnt er een nieuwe afbeelding, en ook iedere keer als we een nieuwe sector lezen. Omdat onze fantoomcursors in de sector-afbeelding komen te staan, beginnen we hier met een aanroep van SCHRIJF_FANTOOM in START_SEC_AFB te zetten. Op die manier kunnen we iedere keer als we een nieuwe sector-afbeelding schrijven de fantoomcursors erbij zetten.

Dit is de herziene — en definitieve — versie van START_SEC_AFB in TOON_SEC.ASM:

Listing 21-1. Veranderingen START_SEC_AFB in TOON_SEC.ASM

```

PUBLIC  START_SEC_AFB
EXTRN  SCHRIJF_PATROON:NEAR, STUUR_CRLF:NEAR
EXTRN  GANAAR_XY:NEAR, SCHRIJF_FANTOOM:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN  REGELS_VOOR_SECTOR:BYTE
EXTRN  SECTOR_OFFSET:WORD
DATA_SEG ENDS

;-----;
; Deze procedure start de halve sector-afbeelding.
;
; Gebruikt:  SCHRIJF_PATROON, STUUR_CRLF, TOON_HALVE_SECTOR
;            SCHRIJF_HEX_GETALLEN_BOVENLANGS, GANAAR_XY,
;            SCHRIJF_FANTOOM
; Leest:    BOVENSTE_REGEL_PATROON, ONDERSTE_REGEL_PATROON
;            REGELS_VOOR_SECTOR
; Schrijft: SECTOR_OFFSET
;-----;
START_SEC_AFB PROC NEAR
    PUSH    DX
    XOR     DL,DL                ;zet cursor op plaats
    MOV     DH,REGELS_VOOR_SECTOR
    CALL    GANAAR_XY
    CALL    SCHRIJF_HEX_GETALLEN_BOVENLANGS
    LEA     DX,BOVENSTE_REGEL_PATROON
    CALL    SCHRIJF_PATROON
    CALL    STUUR_CRLF
    XOR     DX,DX                ;begin bij begin van de sector

```

Listing 21-1. *vervolg*

```

MOV     SECTOR_OFFSET,DX      ;maak sector-offset 0
CALL    TOON_HALVE_SECTOR
LEA     DX,ONDERSTE_REGEL_PATROON
CALL    SCHRIJF_PATROON
CALL    SCHRIJF_FANTOOM      ;schrijf de fantoomcursor
POP     DX
RET
START_SEC_AFB    ENDP

```

Merk op dat we START_SEC_AFB ook zodanig hebben gewijzigd dat hij variabelen gebruikt en initialiseert. Hij maakt nu SECTOR_OFFSET gelijk aan 0 zodat de eerste helft van een sector wordt afgebeeld.

Nu SCHRIJF_FANTOOM zelf. Die vergt nogal wat werk. Alles bij elkaar moeten we zes procedures schrijven, met inbegrip van SCHRIJF_FANTOOM. Het idee is echter vrij eenvoudig. Eerst verplaatsen we de echte cursor naar de positie van de fantoomcursor in het hex-venster en veranderen het attribuut van de volgende vier tekens in inverse video (attribuut 70h). Hierdoor ontstaan een wit blokje, van vier tekens breed, met een zwart hex-getal. Daarna doen we hetzelfde in het ASCII-venster, maar dan voor één enkel teken. Ten slotte zetten we de echte cursor weer terug naar waar hij aanvankelijk stond. Alle procedures voor de fantoomcursor komen in FANTOOM.ASM te staan, behalve SCHRIJF_ATTRIBUUT_N_KEER, de procedure die het attribuut van tekens instelt.

Voer de volgende procedures in het bestand FANTOOM.ASM in.

Listing 21-2. Het nieuwe bestand FANTOOM.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC

        PUBLIC GA_NAAR_HEX_POSITIE
        EXTRN  GANAAR_XY:NEAR
DATA_SEG      SEGMENT PUBLIC
        EXTRN  REGELS_VOOR_SECTOR:BYTE
DATA_SEG      ENDS

;-----;
; Deze procedure zet de echte cursor op de plaats van de fantoomcursor ;
; in het hex-venster. ;
; ;
; Gebruikt:  GANAAR_XY ;
; Leest:     REGELS_VOOR_SECTOR, FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
;-----;
GA_NAAR_HEX_POSITIE  PROC    NEAR
    PUSH    AX
    PUSH    CX
    PUSH    DX
    MOV     DH,REGELS_VOOR_SECTOR    ;zoek rij van fantoom (0,0)
    ADD     DH,2                    ;plus rij van hex- en horizontale balk
    ADD     DH,FANTOOMCURSOR_Y      ;DH = rij van fantoomcursor
    MOV     DL,8                    ;aan linkerkant inspringen

```


Listing 21-2. *vervolg*

```

MOV     CL,3                      ;elke kolom gebruikt 3 tekens, dus
MOV     AL,FANTOOMCURSOR_X        ; CURSOR_X moet met 3 vermenigvuldigd
MUL     CL
ADD     DL,AL                     ;en optellen bij inspringing, om kolom
CALL    GANAAR_XY                 ; voor fantoomcursor te verkrijgen
POP     DX
POP     CX
POP     AX
RET

GA_NAAR_HEX_POSITIE      ENDP

        PUBLIC  GA_NAAR_ASCII_POSITIE
        EXTRN   GANAAR_XY:NEAR
DATA_SEG SEGMENT PUBLIC
        EXTRN   REGELS_VOOR_SECTOR:BYTE
DATA_SEG ENDS

;-----;
; Deze procedure zet de echte cursor aan het begin van de fantoomcursor ;
; in het ASCII-venster. ;
; ;
; Gebruikt:      GANAAR_XY ;
; Leest:         REGELS_VOOR_SECTOR, FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
;-----;
GA_NAAR_ASCII_POSITIE    PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     DH,REGELS_VOOR_SECTOR ;zoek rij van fantoom (0,0)
        ADD     DH,2                  ;plus rij van hex- en horizontale balk
        ADD     DH,FANTOOMCURSOR_Y   ;DH = rij van fantoomcursor
        MOV     DL,59                ;aan linkerkant inspringen
        ADD     DL,FANTOOMCURSOR_X   ;tel CURSOR_X erbij op om positie
        CALL    GANAAR_XY            ; X voor fantoomcursor te verkrijgen
        POP     DX
        POP     AX
        RET

GA_NAAR_ASCII_POSITIE    ENDP

        PUBLIC  BEWAAR_ECHTE_CURSOR
;-----;
; Deze procedure bewaart de plaats van de echte cursor in de twee ;
; variabelen ECHTE_CURSOR_X en ECHTE_CURSOR_Y. ;
; ;
; Schrijft:      ECHTE_CURSOR_X, ECHTE_CURSOR_Y ;
;-----;
BEWAAR_ECHTE_CURSOR      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                  ;lees cursorpositie
        XOR     BH,BH                 ; op pagina 0
        INT     10h                  ;en zet in DL,DH
        MOV     ECHTE_CURSOR_Y,DL    ;bewaar positie
        MOV     ECHTE_CURSOR_X,DH
        POP     DX

```

Listing 21-2. *vervolg*

```

        POP        CX
        POP        BX
        POP        AX
        RET
BEWAAR_ECHTE_CURSOR    ENDP

        PUBLIC    HERSTEL_ECHTE_CURSOR
        EXTRN     GANAAR_XY:NEAR
;-----;
; Deze procedure zet de echte cursor weer op zijn oude plaats, die is ;
; bewaard in ECHTE_CURSOR_X en ECHTE_CURSOR_Y. ;
; ;
; Gebruikt:      GANAAR_XY ;
; Leest:         ECHTE_CURSOR_X, ECHTE_CURSOR_Y ;
;-----;
HERSTEL_ECHTE_CURSOR    PROC    NEAR
        PUSH     DX
        MOV      DL,ECHTE_CURSOR_Y
        MOV      DH,ECHTE_CURSOR_X
        CALL     GANAAR_XY
        POP      DX
        RET
HERSTEL_ECHTE_CURSOR    ENDP

        PUBLIC    SCHRIJF_FANTOOM
        EXTRN     SCHRIJF_ATTRIBUUT_N_KEER:NEAR
;-----;
; Deze procedure gebruikt CURSOR_X en CURSOR_Y, via GA_NAAR..., als de ;
; coördinaten voor de fantoomcursor. SCHRIJF_FANTOOM schrijft deze ;
; fantoomcursor. ;
; ;
; Gebruikt:      SCHRIJF_ATTRIBUUT_N_KEER, BEWAAR_ECHTE_CURSOR ;
;               HERSTEL_ECHTE_CURSOR, GA_NAAR_HEX_POSITIE ;
;               GA_NAAR_ASCII_POSITIE ;
;-----;
SCHRIJF_FANTOOM    PROC    NEAR
        PUSH     CX
        PUSH     DX
        CALL     BEWAAR_ECHTE_CURSOR
        CALL     GA_NAAR_HEX_POSITIE      ;coörd. van cursor in hex-venster
        MOV      CX,4                    ;maak fantoomcursor vier tekens breed
        MOV      DL,70h
        CALL     SCHRIJF_ATTRIBUUT_N_KEER
        CALL     GA_NAAR_ASCII_POSITIE   ;coörd. van cursor in ASCII-venster
        MOV      CX,1                    ;cursor is hier een teken breed
        CALL     SCHRIJF_ATTRIBUUT_N_KEER
        CALL     HERSTEL_ECHTE_CURSOR
        POP      DX
        POP      CX
        RET
SCHRIJF_FANTOOM    ENDP

        PUBLIC    WIS_FANTOOM
        EXTRN     SCHRIJF_ATTRIBUUT_N_KEER:NEAR

```


Listing 21-2. *vervolg*

```

;-----;
; Deze procedure verwijdert de fantoomcursor - precies het
; tegenovergestelde van SCHRIJF_FANTOOM.
;
;
; Gebruikt:   SCHRIJF_ATTRIBUT_N_KEER, BEWAAR_ECHTE_CURSOR
;             HERSTEL_ECHTE_CURSOR, GA_NAAR_HEX_POSITIE
;             GA_NAAR_ASCII_POSITIE
;-----;
WIS_FANTOOM    PROC    NEAR
    PUSH    CX
    PUSH    DX
    CALL    BEWAAR_ECHTE_CURSOR
    CALL    GA_NAAR_HEX_POSITIE    ;coörd. van cursor in hex-venster
    MOV     CX,4                  ;verander weer naar wit op zwart
    MOV     DL,7
    CALL    SCHRIJF_ATTRIBUT_N_KEER
    CALL    GA_NAAR_ASCII_POSITIE
    MOV     CX,1
    CALL    SCHRIJF_ATTRIBUT_N_KEER
    CALL    HERSTEL_ECHTE_CURSOR
    POP     DX
    POP     CX
    RET
WIS_FANTOOM    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
ECHTE_CURSOR_X    DB    0
ECHTE_CURSOR_Y    DB    0
    PUBLIC  FANTOOMCURSOR_X, FANTOOMCURSOR_Y
FANTOOMCURSOR_X    DB    0
FANTOOMCURSOR_Y    DB    0
DATA_SEG      ENDS

END

```

SCHRIJF_FANTOOM en WIS_FANTOOM zijn vrijwel gelijk. Het enige verschil zit in het gebruikte attribuut; SCHRIJF_FANTOOM zet het attribuut op 70h voor inverse video, terwijl WIS_FANTOOM het attribuut weer de normale waarde (7) geeft.

Beide procedures bewaren de oude plaats van de cursor met BEWAAR_ECHTE_CURSOR, die functie 3 van INT 10h gebruikt om de plaats van de cursor te lezen en deze plaats dan in de twee bytes ECHTE_CURSOR_X en ECHTE_CURSOR_Y bewaart.

Na de positie van de echte cursor te hebben bewaard, roepen zowel SCHRIJF_FANTOOM als WIS_FANTOOM daarna GA_NAAR_HEX_POSITIE aan, die de cursor aan het begin van de fantoomcursor in het hex-venster zet. Vervolgens schrijft SCHRIJF_ATTRIBUT_N_KEER het inverse video-attribuut voor vier tekens, vanaf de cursor naar rechts. Dit gebeurt in het hex-venster. Op dezelfde ma-

nier schrijft SCHRIJF_FANTOOM dan een fantoomcursor van één teken breed in het ASCII-venster. Ten slotte zet HERSTEL_ECHTE_CURSOR de echte cursor weer op de plaats waar hij stond voordat SCHRIJF_FANTOOM werd aangeroepen.

De enige procedure die op dit moment nog niet is geschreven, is SCHRIJF_ATTRIBUTUUT_N_KEER, dus laten we dat nu doen.

21.2 Tekenattributen veranderen

We gaan SCHRIJF_ATTRIBUTUUT_N_KEER voor drie dingen gebruiken. Allereerst leest hij het teken op de plaats van de cursor. We doen dat omdat de INT 10h-functie die we gebruiken om het attribuut van een teken in te stellen, nummer 9, zowel het teken als het attribuut onder de cursor schrijft. SCHRIJF_ATTRIBUTUUT_N_KEER zal op die manier het attribuut veranderen door het nieuwe attribuut samen met het juist gelezen teken te schrijven. Ten slotte zet de procedure de cursor op de volgende tekenpositie rechts, zodat we het hele proces N keer kunnen herhalen. De details kunt u in de procedure zelf zien; zet SCHRIJF_ATTRIBUTUUT_N_KEER in het bestand VIDEO_IO.ASM:

Listing 21-3. Voeg deze procedure toe aan VIDEO_IO.ASM.

```

PUBLIC SCHRIJF_ATTRIBUTUUT_N_KEER
EXTRN  CURSOR_RECHTS:NEAR
;-----;
; Deze procedure stelt het attribuut in op N tekens, vanaf de huidige ;
; cursorpositie. ;
; ;
; CX      Aantal tekens waarvoor attribuut moet worden ingesteld ;
; DL      Nieuw attribuut voor tekens ;
; ;
; Gebruikt:      CURSOR_RECHTS ;
;-----;
SCHRIJF_ATTRIBUTUUT_N_KEER      PROC    NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     BL,DL                ;stel attribuut in op nieuw attribuut
    XOR     BH,BH                ;stel scherpagina in op 0
    MOV     DX,CX                ;CX wordt gebruikt door BIOS-routines
    MOV     CX,1                ;stel attribuut voor een teken in
ATTR_LUS:
    MOV     AH,8                ;lees teken onder cursor
    INT     10h
    MOV     AH,9                ;schrijf attribuut/teken
    INT     10h
    CALL    CURSOR_RECHTS
    DEC     DX                  ;attribuut voor N tekens instellen?
    JNZ     ATTR_LUS            ;nee, ga door
    POP     DX
    POP     CX
    POP     BX
    POP     AX

```


Listing 21-3. *vervolg*

```
RET
SCHRIJF_ATTRIBUT_N_KEER      ENDP
```

Dit is meteen de definitieve versie van SCHRIJF_ATTRIBUT_N_KEER. Het betekent ook meteen de definitieve versie van VIDEO_IO.ASM, dus die hoeft u nu niet meer te veranderen of te assembleren.

21.3 Samenvatting

We hebben nu acht bestanden om te linken, met de hoofdprocedure nog steeds in Dskpatch. Hiervan hebben we twee bestanden veranderd, Toon_sec en Video_io, en er één aangemaakt, Fantoom. Als u Make of het batch-bestandje dat we in hoofdstuk 20 gaven, gebruikt, denk er dan aan uw nieuwe bestand, Fantoom, aan de lijst toe te voegen.

Als u Dskpatch nu draait, zult u zien hoe hij, net als voorheen, de sector-afbeelding schrijft, maar Dskpatch zal nu ook de twee fantoomcursors erbij zetten. (Zie afb. 21-1.) Merk op dat de echte cursor helemaal aan het eind weer staat waar hij eerst stond. In het volgende hoofdstuk voegen we procedures toe om onze nieuwe fantoomcursors te verplaatsen, en laten we een eenvoudige procedure zien waarmee we de byte onder de fantoomcursor kunnen wijzigen.

Drive A	Sector 0																0123456789ABCDEF															
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F																
00	13	2A	90	49	42	4D	20	20	33	2E	30	00	02	02	01	00	5*	E	I	B	M	3.0	00	00								
10	02	70	00	D0	02	FD	02	00	09	00	02	00	00	00	00	00	0p	40	20	0	0											
20	14	00	00	00	00	0F	00	00	00	00	00	00	FA	33	C0	8E	0	*														
30	D0	BC	00	7C	BB	78	00	36	C5	37	1E	56	16	53	8E	C0	0															
40	BF	21	7C	B9	0B	00	FC	AC	26	80	3D	00	74	03	26	8A	0															
50	05	AA	E2	F3	8C	C0	8E	D8	89	47	02	C7	07	21	7C	FB	0															
60	CD	13	73	03	E9	C7	00	0E	1F	A0	10	7C	98	F7	26	16	0															
70	7C	03	06	1C	7C	03	06	0E	7C	A3	2C	7C	A3	2E	7C	B8	0															
80	20	00	F7	26	11	7C	BB	00	02	03	C3	48	F7	F3	01	06	0															
90	2E	7C	B8	50	00	8E	C0	A1	2C	7C	E8	A7	00	33	DB	B8	0															
A0	01	02	E8	B9	00	72	1A	33	FF	B9	0B	00	BE	D6	7D	FC	0															
B0	F3	A6	75	0D	BF	20	00	BE	E1	7D	B9	0B	00	F3	A6	74	0															
C0	13	BE	77	7D	E8	6F	00	32	E4	CD	16	5E	1F	8F	04	8F	0															
D0	44	02	CD	19	26	A1	1C	00	BB	00	02	33	D2	F7	F3	FE	0															
E0	C0	A2	20	7C	A1	2E	7C	A3	75	7D	B8	70	00	8E	C0	33	0															
F0	DB	A1	2E	7C	E8	4D	00	A0	18	7C	2A	06	16	7C	FE	C0	0															

Druk op functietoets, of voer teken of hex-byte in:

Afb. 21-1. Afbeelding met fantoomcursors.

22 Eenvoudige wijzigingen

- 22.1 De fantoomcursors verplaatsen 234**
- 22.2 Eenvoudige wijzigingen 236**
- 22.3 Samenvatting 240**

We zijn bijna op het punt waarop we onze sector-afbeelding kunnen gaan 'editen': er getallen in veranderen. We voegen straks eenvoudige versies van de procedure voor het wijzigen van bytes in onze afbeelding toe, maar eerst moeten we een manier vinden om de fantoomcursors naar verschillende bytes binnen de halve sector-afbeelding te verplaatsen. Dit blijkt vrij eenvoudig te zijn nu we beschikken over de twee procedures WIS__FANTOOM en SCHRIJF__FANTOOM.

22.1 De fantoomcursors verplaatsen

Het verplaatsen van de fantoomcursors in elke richting is gebaseerd op drie basistappen: de fantoomcursor op zijn huidige positie wissen, de cursorpositie wijzigen door een van de variabelen FANTOOMCURSOR__X of FANTOOMCURSOR__Y te wijzigen, en de fantoomcursor met SCHRIJF__FANTOOM naar de nieuwe positie schrijven. Daarbij moeten we echter oppassen dat de cursor niet buiten het venster komt, dat 16 bytes breed en 16 bytes hoog is.

Om de fantoomcursors te verplaatsen, hebben we vier nieuwe procedures nodig, één voor elke pijltoets op het toetsenbord. VERDELER hoeft niet te worden veranderd, omdat alle informatie over procedures en uitgebreide codes in de tabel VERDEEL__TABEL staat. We hoeven er alleen de uitgebreide ASCII-codes en adressen van de procedures voor elk van de pijltoetsen aan toe te voegen. Dit zijn de toevoegingen aan VERDEEL.ASM waardoor de cursortoetsen tot leven komen:

Listing 22-1. Veranderingen in VERDEEL.ASM

```
DATA_SEG      SEGMENT PUBLIC

CODE_SEG      SEGMENT PUBLIC
    EXTRN      VOLGENDE_SECTOR:NEAR          ;in DISK_IO.ASM
    EXTRN      VORIGE_SECTOR:NEAR           ;in DISK_IO.ASM
    EXTRN      FANTOOM_OMHOOG:NEAR, FANTOOM_OMLAAG:NEAR;in FANTOOM.ASM
    EXTRN      FANTOOM_LINKS:NEAR, FANTOOM_RECHTS:NEAR
CODE_SEG      ENDS

;-----;
; Deze tabel bevat de toegestane uitgebreide ASCII-toetsen en de ;
; adressen van de procedures die moeten worden aangeroepen wanneer een ;
; toets wordt ingedrukt. ;
; De tabel heeft de volgende indeling: ;
; DB 72 ;uitgebreide code voor cursor omhoog ;
; DW OFFSET CGROEP:FANTOOM_OMHOOG ;
;-----;
VERDEEL_TABEL LABEL BYTE
    DB 59 ;F1
    DW OFFSET CGROEP:VORIGE_SECTOR
    DB 60 ;F2
    DW OFFSET CGROEP:VOLGENDE_SECTOR
    DB 72 ;cursor omhoog
    DW OFFSET CGROEP:FANTOOM_OMHOOG
    DB 80 ;cursor omlaag
    DW OFFSET CGROEP:FANTOOM_OMLAAG
    DB 75 ;cursor naar links
    DW OFFSET CGROEP:FANTOOM_LINKS
    DB 77 ;cursor naar rechts
```

Listing 22-1. *vervolg*

```

      DW      OFFSET CGROEP:FANTOOM_RECHTS
      DB      0                                ;einde van de tabel
DATA_SEG      ENDS

```

Zoals u ziet is het niet moeilijk om opdrachten aan Dskpatch toe te voegen: we zetten gewoon de procedurenamen in VERDEEL_TABEL en schrijven de procedures. Over het schrijven van procedures gesproken: de procedures FANTOOM_OMHOOG, FANTOOM_OMLAAG, enzovoort, zijn vrij eenvoudig. Ze lijken ook veel op elkaar, en verschillen alleen ten aanzien van de grenscondities die voor elk ervan gelden. We hebben al beschreven hoe ze werken; kijk of u ze zelf kunt schrijven en in het FANTOOM.ASM-bestand zetten voor u verder leest. Onze versies van de procedures voor het verplaatsen van de fantoomcursors zien er zo uit:

Listing 22-2. Voeg deze procedures toe aan FANTOOM.ASM

```

;-----;
; Deze vier procedures verplaatsen de fantoomcursors.                ;
;                                                                    ;
; Gebruikt:   WIS_FANTOOM, SCHRIJF_FANTOOM                            ;
; Leest:      FANTOOMCURSOR_X, FANTOOMCURSOR_Y                        ;
; Schrijft:   FANTOOMCURSOR_X, FANTOOMCURSOR_Y                        ;
;-----;

      PUBLIC  FANTOOM_OMHOOG
FANTOOM_OMHOOG  PROC  NEAR
      CALL   WIS_FANTOOM                ;wis op huidige positie
      DEC    FANTOOMCURSOR_Y            ;zet cursor een regel hoger
      JNS    STOND_NIET_BOVEN           ;stond niet bovenaan, schrijf cursor
      MOV     FANTOOMCURSOR_Y,0         ;stond bovenaan, dus zet daar terug
STOND_NIET_BOVEN:
      CALL   SCHRIJF_FANTOOM            ;schrijf fantoom op nieuwe positie
      RET
FANTOOM_OMHOOG  ENDP

      PUBLIC  FANTOOM_OMLAAG
FANTOOM_OMLAAG  PROC  NEAR
      CALL   WIS_FANTOOM                ;wis op huidige positie
      INC     FANTOOMCURSOR_Y            ;zet cursor een regel lager
      CMP     FANTOOMCURSOR_Y,16         ;stond hij onderaan?
      JB      STOND_NIET_ONDER           ;nee, dus schrijf fantoom
      MOV     FANTOOMCURSOR_Y,15         ;stond onderaan, dus zet daar terug
STOND_NIET_ONDER:
      CALL   SCHRIJF_FANTOOM            ;schrijf de fantoomcursor
      RET
FANTOOM_OMLAAG  ENDP

      PUBLIC  FANTOOM_LINKS
FANTOOM_LINKS   PROC  NEAR
      CALL   WIS_FANTOOM                ;wis op huidige positie
      DEC     FANTOOMCURSOR_X            ;zet cursor een kolom naar links
      JNS     STOND_NIET_LINKS           ;stond niet links, schrijf cursor
      MOV     FANTOOMCURSOR_X,0         ;stond links, dus zet daar terug

```


Listing 22-2. *vervolg*

```

STOND_NIET_LINKS:
    CALL    SCHRIJF_FANTOOM          ;schrijf de fantoomcursor
    RET
FANTOOM_LINKS    ENDP
PUBLIC FANTOOM_RECHTS
FANTOOM_RECHTS   PROC    NEAR
    CALL    WIS_FANTOOM              ;wis op huidige positie
    INC     FANTOOMCURSOR_X          ;zet cursor een kolom naar rechts
    CMP     FANTOOMCURSOR_X,16       ;stond hij al aan rechterkant?
    JB      STOND_NIET_RECHTS
    MOV     FANTOOMCURSOR_X,15       ;stond rechts, dus zet daar terug
STOND_NIET_RECHTS:
    CALL    SCHRIJF_FANTOOM          ;schrijf de fantoomcursor
    RET
FANTOOM_RECHTS   ENDP

```

FANTOOM__LINKS en FANTOOM__RECHTS zijn de definitieve versies, maar we zullen FANTOOM__OMHOOG en FANTOOM__OMLAAG nog moeten wijzigen wanneer we de schermafbeelding gaan 'scrollen'.

In de huidige vorm van Dskpatch kunnen we alleen de eerste helft van een sector zien. In hoofdstuk 27 zullen we nog wat dingen aan Dskpatch toevoegen en erin veranderen om de afbeelding zo te kunnen verschuiven dat we ook andere delen van de sector kunnen zien. We zullen dan zowel FANTOOM__OMHOOG als FANTOOM__OMLAAG zo veranderen dat ze het scherm scrollen wanneer we de cursor voorbij de boven- of onderkant van het scherm proberen te verplaatsen. Als de cursor dan bijvoorbeeld onderin de halve sector-afbeelding staat, moet een druk op de cursor omlaag-toets tot gevolg hebben dat de afbeelding een regel naar boven schuift en er een regel aan de onderkant bij komt te staan zodat we de volgende 16 bytes zien. Het verschuiven van zo'n afbeelding is echter een nogal lastige kwestie, dus we zullen de desbetreffende procedure tot bijna het laatst bewaren. Tot en met hoofdstuk 26 zullen we het deel van Dskpatch dat zich bezighoudt met het editen van de afbeelding en de invoer via het toetsenbord ontwikkelen door alleen de eerste halve sector te gebruiken.

Test Dskpatch nu om te zien of u de fantoomcursors kunt verplaatsen over het scherm. Ze moeten tegelijk bewegen, en binnen hun eigen venster blijven. Nu gaan we de procedures voor het wijzigen van bytes in de afbeelding toevoegen.

22.2 Eenvoudige wijzigingen

We hebben al een eenvoudige toetsenbordinvoer-procedure, LEES__BYTE, die één teken van het toetsenbord leest zonder te wachten tot u de Enter-toets indrukt. We gaan deze oude testversie van LEES__BYTE gebruiken voor het aanbrengen van wijzigingen. In het volgende hoofdstuk schrijven we dan een verfijndere versie van deze edit-procedure die wacht tot we hetzij de Enter-toets hetzij een speciale toets, zoals een functietoets of cursortoets, hebben ingedrukt.

We noemen onze wijzigingsprocedure EDIT__BYTE en hij zal een byte zowel op het scherm als in het geheugen (SECTOR) veranderen. EDIT__BYTE pakt het teken in het DL-register, schrijft het naar de geheugenplaats binnen SECTOR die op dat mo-

ment door de fantoomcursor wordt aangewezen en verandert dan de afbeelding. VERDELER heeft al een leuk plekje van waaruit we EDIT_BYTE kunnen aanroepen. Dit is de nieuwe versie van VERDELER in VERDEEL.ASM, met de aanroep van EDIT_BYTE en de bijbehorende veranderingen:

Listing 22-3. Veranderingen VERDELER in VERDEEL.ASM

```

PUBLIC VERDELER
EXTRN LEES_BYTE:NEAR, EDIT_BYTE:NEAR
;-----;
; Dit is de centrale verdeler. Bij normaal wijzigen (editen) en ;
; bekijken, leest deze procedure tekens van het toetsenbord en, als het ;
; teken een opdrachttoets (b.v. een cursortoets) is, roept VERDELER ;
; procedures aan die het eigenlijke werk doen. Dit gebeurt bij speciale ;
; toetsen die vermeld staan in de tabel VERDEEL_TABEL, waarin de ;
; adressen van de procedures vlak na de toetsnamen zijn opgeslagen. ;
; Als het teken geen speciale toets is, moet het rechtstreeks in de ;
; sectorbuffer worden gezet -- dit is de edit-modus. ;
; ;
; Gebruikt: LEES_BYTE, EDIT_BYTE ;
;-----;
VERDELER PROC NEAR
    PUSH AX
    PUSH BX
    PUSH DX
VERDEEL_LUS:
    CALL LEES_BYTE ;lees teken naar AX
    OR AH,AH ;AX = 0 als geen teken gelezen, -1
    ; voor code uitgebreide tekenset
    JZ VERDEEL_LUS ;geen teken gelezen, probeer opnieuw
    JS SPECIALE_TOETS ;lees uitgebreide code
;doe voorlopig niets met het teken
    MOV DL,AL
    CALL EDIT_BYTE ;was normaal teken, wijzig byte
    JMP VERDEEL_LUS ;lees nog een teken

SPECIALE_TOETS:
    CMP AL,68 ;F10--stop?
    JE EINDE_VERDEEL ;ja, einde
    LEA BX,VERDEEL_TABEL ;gebruik BX om tabel door te kijken

SPECIALE_LUS:
    CMP BYTE PTR [BX],0 ;einde van tabel?
    JE NIET_IN_TABEL ;ja, toets stond niet in tabel
    CMP AL,[BX] ;is het deze tabel-ingang?
    JE VERDEEL ;ja, dan uitvoeren
    ADD BX,3 ;nee, probeer volgende ingang
    JMP SPECIALE_LUS ;controleer volgende tabel-ingang

VERDEEL:
    INC BX ;wijs naar adres van procedure
    CALL WORD PTR [BX] ;roep procedure aan
    JMP VERDEEL_LUS ;wacht op een andere toets

NIET_IN_TABEL:
    JMP VERDEEL_LUS ;doe niets, lees gewoon volgende teken

```


Listing 22-3. *vervolg*

```

EINDE_VERDEEL:
    POP     DX
    POP     BX
    POP     AX
    RET
VERDELER      ENDP

```

De EDIT_BYTE-procedure doet zelf een hoop werk dat bijna uitsluitend bestaat uit het aanroepen van andere procedures, wat een van de kenmerken van modulair ontwerpen is. Bij een modulaire opbouw kunnen we vaak vrij ingewikkelde procedures schrijven door gewoon een reeks aanroepen van andere procedures te doen die vervolgens het werk verrichten. Veel procedures in EDIT_BYTE werken met een teken in het DL-register, maar dat is al ingesteld wanneer we EDIT_BYTE aanroepen, dus de enige andere instructie dan een CALL (of PUSH, POP) is de LEA-instructie die het adres van de prompt instelt voor SCHRIJF_PROMPTREGEL. De meeste procedure-aanroepen in EDIT_BYTE zijn voor het bijwerken van de afbeelding wanneer we een byte hebben gewijzigd. De overige details van EDIT_BYTE ziet u wanneer we straks bij de listing van de procedure zijn aangekomen.

Omdat EDIT_BYTE de byte op het scherm verandert, hebben we nóg een procedure, SCHRIJF_NAAR_GEHEUGEN nodig om de byte in SECTOR te veranderen. SCHRIJF_NAAR_GEHEUGEN gebruikt de coördinaten in FANTOOMCURSOR_X en FANTOOMCURSOR_Y voor het berekenen van de offset van de fantoomcursor in SECTOR, schrijft dan het teken (de byte) in het DL-register naar de juiste byte binnen SECTOR.

Hier is het nieuwe bestand EDITOR.ASM, dat de uiteindelijke versies van zowel EDIT_BYTE als SCHRIJF_NAAR_GEHEUGEN bevat.

Listing 22-4. Het nieuwe bestand EDITOR.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC

DATA_SEG      SEGMENT PUBLIC
        EXTRN  SECTOR:BYTE
        EXTRN  SECTOR_OFFSET:WORD
        EXTRN  FANTOOMCURSOR_X:BYTE
        EXTRN  FANTOOMCURSOR_Y:BYTE
DATA_SEG      ENDS

```

Listing 22-4. *vervolg*

```

;-----;
; Deze procedure schrijft een byte naar SECTOR, op de geheugenplaats ;
; die door de fantoomcursors wordt aangewezen. ;
; ;
; DL Naar SECTOR te schrijven byte ;
; ;
; De offset wordt berekend met ;
; OFFSET = SECTOR_OFFSET + (16 * FANTOOMCURSOR_Y) + FANTOOMCURSOR_X ;
; ;
; Leest: FANTOOMCURSOR_X, FANTOOMCURSOR_Y, SECTOR_OFFSET ;
; Schrijft: SECTOR ;
;-----;
SCHRIJF_NAAR_GEHEUGEN PROC NEAR
    PUSH AX
    PUSH BX
    PUSH CX
    MOV BX,SECTOR_OFFSET
    MOV AL,FANTOOMCURSOR_Y
    XOR AH,AH
    MOV CL,4 ;vermenigvuldig FANTOOMCURSOR_Y met 16
    SHL AX,CL
    ADD BX,AX ;BX = SECTOR_OFFSET + (16 * Y)
    MOV AL,FANTOOMCURSOR_X
    XOR AH,AH
    ADD BX,AX ;dat is het adres!
    MOV SECTOR[BX],DL ;sla nu de byte op
    POP CX
    POP BX
    POP AX
    RET
SCHRIJF_NAAR_GEHEUGEN ENDP

PUBLIC EDIT_BYTE
EXTRN BEWAAR_ECHTE_CURSOR:NEAR, HERSTEL_ECHTE_CURSOR:NEAR
EXTRN GA_NAAR_HEX_POSITIE:NEAR, GA_NAAR_ASCII_POSITIE:NEAR
EXTRN SCHRIJF_FANTOOM:NEAR, SCHRIJF_PROMPTREGEL:NEAR
EXTRN CURSOR_RECHTS:NEAR, SCHRIJF_HEX:NEAR, SCHRIJF_TEK:NEAR
DATA_SEG SEGMENT PUBLIC
    EXTRN EDITOR_PROMPT:BYTE
DATA_SEG ENDS

;-----;
; Deze procedure verandert een byte in het geheugen en op het scherm. ;
; ;
; DL Byte die in SECTOR moet worden geschreven en op scherm ;
; veranderd ;
; ;
; Gebruikt: BEWAAR_ECHTE_CURSOR, HERSTEL_ECHTE_CURSOR ;
; GA_NAAR_HEX_POSITIE, GA_NAAR_ASCII_POSITIE ;
; SCHRIJF_FANTOOM, SCHRIJF_PROMPTREGEL, CURSOR_RECHTS ;
; SCHRIJF_HEX, SCHRIJF_TEK, SCHRIJF_NAAR_GEHEUGEN ;
; Leest: EDITOR_PROMPT ;
;-----;
EDIT_BYTE PROC NEAR
    PUSH DX
    CALL BEWAAR_ECHTE_CURSOR

```


Listing 22-4. *vervolg*

```
CALL    GA_NAAR_HEX_POSITIE    ;ga naar het hex-getal in het
CALL    CURSOR_RECHTS          ; hex-venster
CALL    SCHRIJF_HEX            ;schrijf het hex-getal
CALL    GA_NAAR_ASCII_POSITIE  ;ga naar het teken in het ASCII-venster
CALL    SCHRIJF_TEK            ;schrijf het nieuwe teken
CALL    HERSTEL_ECHTE_CURSOR   ;zet cursor weer waar hij hoort
CALL    SCHRIJF_FANTOOM        ;herschrijf de fantoomcursor
CALL    SCHRIJF_NAAR_GEHEUGEN  ;bewaar deze nieuwe byte in SECTOR
LEA      DX,EDITOR_PROMPT
CALL    SCHRIJF_PROMPTREGEL
POP      DX
RET
EDIT_BYTE    ENDP
CODE_SEG     ENDS

END
```

22.3 Samenvatting

Dskpatch bestaat nu uit negen bestanden: Dskpatch, Verdeel, Toon__sec, Disk__io, Video__io, Tbd__io, Fantoom, Cursor en Editor. In dit hoofdstuk hebben we Verdeel gewijzigd en Editor toegevoegd. Geen van deze bestanden is erg lang, dus het kost niet veel tijd om ze te assembleren. Bovendien kunnen we vrij snel veranderingen aanbrengen door gewoon een van die bestanden te wijzigen, opnieuw te assembleren en dan alle bestanden opnieuw te linken.

In de huidige versie van Dskpatch kunt u nu elke toets indrukken en dan het getal en teken onder de fantoomcursor zien veranderen. We kunnen nu wijzigingen in de afbeelding aanbrengen, maar het is nog niet helemaal veilig omdat we een byte kunnen veranderen door op wat voor toets dan ook te drukken. We moeten nog een soort veiligheid inbouwen, zoals het indrukken van Enter wanneer we een byte veranderd willen hebben, zodat we niet iets veranderen wanneer we per ongeluk op het toetsenbord leunen.

Bovendien kunnen we met de huidige versie van LEES__BYTE geen hex-getal invoeren om een byte te veranderen. In hoofdstuk 24 herschrijven we LEES__BYTE, zowel om een Enter-toets te moeten indrukken voor hij een nieuw teken accepteert als om een hex-getal van twee cijfers te kunnen invoeren. Eerst moeten we daarvoor een hex-invoerprocedure schrijven; in het volgende hoofdstuk schrijven we invoerprocedures voor zowel hex als decimaal.

23 Invoer hex en decimaal

23.1 Hex-invoer 242

23.2 Decimale invoer 248

23.3 Samenvatting 251

In dit hoofdstuk komen we twee nieuwe procedures voor invoer via het toetsenbord tegen: een voor het lezen van een byte in de vorm van hetzij een hex-getal van twee cijfers hetzij een enkel teken, en een voor het lezen van een woord in de vorm van de tekens van een decimaal getal. Dat worden onze hex- en decimale invoerprocedures.

Beide procedures zijn wel zo lastig dat we er een testprogramma voor nodig hebben voor we er zelfs maar over kunnen denken om ze te linken met Dskpatch. We zullen werken met LEES_BYTE, waarvoor een testprocedure van bijzonder belang is omdat deze procedure (tijdelijk) zijn vermogen verliest om speciale functietoetsen te lezen. Omdat Dskpatch van de functietoetsen afhankelijk is, zullen we onze nieuwe LEES_BYTE niet samen met Dskpatch kunnen gebruiken. We zullen ook zien waarom we geen speciale functietoetsen kunnen gebruiken bij de LEES_BYTE die hier wordt ontwikkeld, en in het volgende hoofdstuk zullen we het bestand zodanig wijzigen dat onze functietoets-problemen worden verholpen.

23.1 Hex-invoer

Laten we eerst LEES_BYTE herschrijven. In het vorige hoofdstuk las LEES_BYTE een gewoon teken of een speciale functietoets en gaf een byte door aan Verdeel. Verdeel riep dan de Editor aan als LEES_BYTE een gewoon teken las, en EDIT_BYTE wijzigde dan de byte waarnaar de fantoomcursor wees. Anders zocht Verdeel speciale functietoetsen in VERDEEL_TABEL op om te zien of de byte daar stond; als dat zo was, riep Verdeel de procedure aan die in de tabel werd genoemd.

Maar, zoals we in het vorige hoofdstuk zeiden, de oude versie van LEES_BYTE maakt het veel te gemakkelijk om per ongeluk een byte te wijzigen. Als u onverhoopt een toets van het toetsenbord indrukt die geen speciale toets is, zal EDIT_BYTE de byte onder de fantoomcursor veranderen. We zijn allemaal wel eens onhandig, en een dergelijke onverhoopte wijziging in een sector kan rampen veroorzaken.

We zullen LEES_BYTE zodanig veranderen dat hij voortaan het teken dat we tikken pas doorgeeft als we de Enter-toets hebben ingedrukt. We gaan dit mogelijk maken met de INT 21h-functie 0Ah van DOS, die een tekenstring leest. DOS geeft deze string pas door nadat we Enter hebben ingedrukt, en biedt zo een mooie garantie tegen onhandigheden. We raken daarbij wel onze speciale functietoetsen kwijt, om redenen die u straks zult zien.

Om precies te zien hoe onze veranderingen van invloed zijn op LEES_BYTE, moeten we een testprogramma schrijven om LEES_BYTE afzonderlijk te kunnen testen. Als er dan iets vreemds gebeurt, weten we dat het aan LEES_BYTE ligt en niet aan een ander deel van Dskpatch. We kunnen het ons bij het schrijven van een testprocedure gemakkelijker maken als we een paar procedures van Tbd_io, Video_io en Cursor gebruiken om informatie over de voortgang van LEES_BYTE af te drukken. We zullen procedures als SCHRIJF_HEX en SCHRIJF_DECIMAAL gebruiken om de doorgegeven tekencode en het aantal gelezen tekens te laten zien. De details staan hieronder in TEST.ASM:

Listing 23-1. Het testprogramma TEST.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP
CODE_SEG
        ORG     100h

        EXTRN   SCHRIJF_HEX:NEAR, SCHRIJF_DECIMAAL:NEAR
        EXTRN   SCHRIJF_STRING:NEAR, STUUR_CRLF:NEAR
        EXTRN   LEES_BYTE:NEAR

TEST    PROC    NEAR
        LEA     DX, INVOER_PROMPT
        CALL    SCHRIJF_STRING
        CALL    LEES_BYTE
        CALL    STUUR_CRLF
        LEA     DX, TEKEN_PROMPT
        CALL    SCHRIJF_STRING
        MOV     DL, AL
        CALL    SCHRIJF_HEX
        CALL    STUUR_CRLF
        LEA     DX, TEKENS_GELEZEN_PROMPT
        CALL    SCHRIJF_STRING
        MOV     DL, AH
        XOR     DH, DH
        CALL    SCHRIJF_DECIMAAL
        CALL    STUUR_CRLF
        INT     20h
TEST    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
INVOER_PROMPT DB      'Voer tekens in: ', 0
TEKEN_PROMPT  DB      'Code teken: ', 0
TEKENS_GELEZEN_PROMPT DB 'Aantal gelezen tekens: ', 0
; en nu dummy variabelen
        PUBLIC  KOPREGEL_NR, DISKDRIVE_NR, KOP_DEEL_1, KOP_DEEL_2
        PUBLIC  PROMPTREGEL_NR, HUIDIGE_SECTORNR
KOPREGEL_NR   DB      0
DISKDRIVE_NR  DB      0
KOP_DEEL_1    DB      0
KOP_DEEL_2    DB      0
PROMPTREGEL_NR DB      0
HUIDIGE_SECTORNR DB      0
DATA_SEG      ENDS

        END     TEST

```

Probeer dit eens met uw huidige versie van Tbd_io, Video_io en Cursor te linken (zet Test vooraan in de LINK-lijst). Als u een speciale functietoets indrukt, zal Test zeggen dat hij 255 tekens heeft gelezen. Waarom? We hebben de -1 van AH in DL gezet en de hoge byte van DX 0 gemaakt, waardoor DX bij terugkeer de waarde 255 (FFh), en niet -1 (FFFFh) had.

Als we LEES_BYTE in Dskpatch echt gebruiken, zullen we niet zo onzorgvuldig te werk gaan. Dit is een testprogramma, en zo lang we weten wat we verwachten,

kunnen we LEES_BYTE en al zijn grenscondities testen. Vóór we echter verder gaan met het herschrijven van LEES_BYTE, moeten we nog een onderdeel van TEST.ASM verklaren dat u misschien is opgevallen: de variabele-definities.

Het merendeel van de instructies in TEST.ASM zijn voor de opmaak van het scherm — hoe de afbeelding er mooi kan uitzien. De definities van de variabelen aan het einde van Test zijn er alleen ingezet om de linker tevreden te stellen. Wanneer we Test linken met Tbd_io, Video_io en Cursor, zoekt de linker een aantal variabelen dat wordt gebruikt door Tbd_io, Video_io en Cursor. We hebben die variabelen in Dskpatch gedefinieerd, maar omdat we niet in Dskpatch linken, moeten we deze variabelen opnieuw in TEST.ASM definiëren. We zullen deze variabelen niet echt gebruiken, omdat we geen procedures in Video_io en Cursor aanroepen die ze nodig hebben. Maar we hebben ze wel nodig om de linker duidelijk te maken dat alles er is wat er moet zijn.

We gaan nu LEES_BYTE zo herschrijven dat hij een tekenstring accepteert. Niet alleen kan ons dat ellende besparen wanneer we met Dskpatch een onhandigheid be- gaan, maar we kunnen daarna ook de terugsteltoets (backspace) gebruiken om te- kens te wissen als we ons bij het tikken bedenken of fouten maken — een prettig idee omdat dat nogal eens voorkomt. LEES_BYTE gebruikt de procedure LEES_STRING om een reeks tekens te lezen.

LEES_STRING is een heel eenvoudige, bijna triviale procedure, maar we hebben hem apart gezet om hem in het volgende hoofdstuk te kunnen herschrijven en er dan speciale functietoetsen mee te lezen zonder de Enter-toets te moeten indrukken. Om tijd te besparen, voegen we er nog drie andere procedures aan toe die LEES_BYTE gebruikt: STRING_NAAR_HOOFDLET, CONVERTEER_HEX_CIJFER en HEX_NAAR_BYTE.

STRING_NAAR_HOOFDLET en HEX_NAAR_BYTE werken beide met strings. STRING_NAAR_HOOFDLET zet alle kleine letters in een string om in hoofdletters. Dat betekent dat u zowel f3 als F3 kunt tikken voor het hex-getal F3h. Door de mogelijkheid om hex-getallen zowel met kleine als met hoofdletters te tik- ken, maken we Dskpatch gebruikersvriendelijker.

HEX_NAAR_BYTE pakt de door DOS gelezen string op nadat we STRING_NAAR_HOOFDLET hebben aangeroepen en zet de tweecijferige hex- string om in een getal van één byte. HEX_NAAR_BYTE gebruikt CONVER- TEER_HEX_CIJFER om elk hex-cijfer om te zetten in een getal van vier bits. Hoe zorgen we nu dat DOS niet meer dan twee hex-cijfers leest? De DOS-functie 0Ah leest een hele tekenstring in een geheugengebied dat als volgt is gedefinieerd:

GRENS_AANTAL_TEK	DB	0
GELEZEN_AANTAL_TEK	DB	0
STRING	DB	80 DUP (0)

De eerste byte zorgt dat we niet te veel tekens lezen. GRENS_AANTAL_TEK ver- tellt DOS hoe veel tekens het, hoogstens, moet lezen. Als we hem op drie zetten, leest DOS tot aan twee tekens, plus het teken voor de carriage return (dat DOS altijd mee- tellt). Alle tekens die we daarna intikken, worden genegeerd — weggegooid — en bij elk extra teken zal DOS een pieptoon geven om ons te laten weten dat we de grens gepasseerd zijn. Wanneer we de Enter-toets indrukken, zet DOS de tweede byte, GELEZEN_AANTAL_TEK, op het aantal tekens dat het feitelijk heeft gelezen,

zonder de carriage return mee te tellen.

STRING__NAAR__HOOFDLET en LEES__BYTE gebruiken beide GELEZEN__AANTAL__TEK. LEES__BYTE gebruikt GELEZEN__AANTAL__TEK bijvoorbeeld om na te gaan of u een enkel teken of een hex-getal van twee cijfers hebt getikt. Als GELEZEN__AANTAL__TEK gelijk aan 1 is, zet LEES__BYTE één teken in het AL-register. Als GELEZEN__AANTAL__TEK op twee staat, zet LEES__BYTE de tweecijferige hex-string met HEX__NAAR__BYTE om in één byte.

Hier is het nieuwe bestand TBD_IO.ASM, met alle vier nieuwe procedures:

Listing 23-2. De nieuwe versie van TBD_IO.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC

        PUBLIC STRING_NAAR_HOOFDLET
;-----;
; Deze procedure zet de string, met het DOS-formaat voor strings, om in ;
; een reeks hoofdletters. ;
; ;
;      DS:DX  Adres van stringbuffer ;
;-----;
STRING_NAAR_HOOFDLET      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                      ;wijs naar aantal tekens
        MOV     CL,[BX]                ;aantal tekens in 2de byte van buffer
        XOR     CH,CH                  ;wis hoge byte van teken-telling
HOOFDLET_LUS:
        INC     BX                      ;wijs naar volgende teken in buffer
        MOV     AL,[BX]
        CMP     AL,'a'                  ;kijk of het een kleine letter is
        JB      NIET_KLEIN              ;nee
        CMP     AL,'z'
        JA      NIET_KLEIN
        ADD     AL,'A'-'a'              ;zet om in hoofdletter
        MOV     [BX],AL
NIET_KLEIN:
        LOOP    HOOFDLET_LUS
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_NAAR_HOOFDLET      ENDP
```


Listing 23-2. *vervolg*

```

;-----;
; Deze procedure zet een teken van ASCII (hex) om in een nibble
; (4 bits).
;
;
;      AL      Om te zetten teken
; Geeft door: AL      Nibble
;      DF      Ingesteld bij fout, anders nul
;-----;
CONVERTEER_HEX_CIJFER  PROC    NEAR
    CMP        AL,'0'                ;is 't een geldig cijfer?
    JB         ONGELDIG_CIJFER       ;nee
    CMP        AL,'9'                ;nog niet zeker
    JA         PROBEER_HEX           ;is misschien hex-cijfer
    SUB        AL,'0'                ;is decimaal cijfer, zet om in nibble
    CLC                          ;zet carry uit, geen fout
    RET
PROBEER_HEX:
    CMP        AL,'A'                ;nog niet zeker
    JB         ONGELDIG_CIJFER       ;geen hex
    CMP        AL,'F'                ;nog niet zeker
    JA         ONGELDIG_CIJFER       ;geen hex
    SUB        AL,'A'-10              ;is hex, zet om in nibble
    CLC                          ;zet carry uit, geen fout
    RET
ONGELDIG_CIJFER:
    STC                          ;zet carry aan, fout
    RET
CONVERTEER_HEX_CIJFER  ENDP

PUBLIC  HEX_NAAR_BYTE

;-----;
; Deze procedure zet de twee tekens in DS:DX van hex om in een byte.
;
;      DS:DX    Adres van twee tekens voor hex-getal
; Geeft:
;      AL      Byte
;      CF      Bij fout aan, uit als geen fout
;
; Gebruikt:    CONVERTEER_HEX_CIJFER
;-----;
HEX_NAAR_BYTE  PROC    NEAR
    PUSH        BX
    PUSH        CX
    MOV         BX,DX                ;zet adres in BX voor indirect adres
    MOV         AL,[BX]              ;haal eerste cijfer
    CALL        CONVERTEER_HEX_CIJFER
    JC          ONGELDIG_HEX         ;onjuist hex-cijfer als carry aan
    MOV         CX,4                 ;nu vermenigvuldigen met 16
    SHL         AL,CL
    MOV         AH,AL                ;bewaar een kopie
    INC         BX                   ;haal tweede cijfer
    MOV         AL,[BX]
    CALL        CONVERTEER_HEX_CIJFER
    JC          ONGELDIG_HEX         ;onjuist hex-cijfer als carry aan
    OR          AL,AH                ;combineer twee nibbles

```

Listing 23-2. *vervolg*

```

        CLC                                ;zet carry uit als geen fout
KLAAR_HEX:
        POP     CX
        POP     BX
        RET
ONGELDIG_HEX:
        STC                                ;zet carry aan bij fout
        JMP     KLAAR_HEX
HEX_NAAR_BYTE ENDP

;-----;
; Dit is een eenvoudige versie van LEES_STRING.
;
;      DS:DX  Adres van string-gebied
;-----;
LEES_STRING PROC NEAR
        PUSH    AX
        MOV     AH,0Ah                    ;aanroep gebufferde toetsenbord-invoer
        INT     21h                      ;aanroep DOS-functie voor gebufferde
                                         ; invoer
        POP     AX
        RET
LEES_STRING ENDP

        PUBLIC  LEES_BYTE
;-----;
; Deze procedure leest een enkel ASCII-teken of een hex-getal
; van twee cijfers. Dit is nog maar een testversie van LEES_BYTE.
;
; Zet byte in      AL      Code teken (tenzij AH = 0)
;                  AH      1 als ASCII-teken gelezen
;                  0 als geen tekens gelezen
;                  -1 als een speciale toets gelezen
;
; Gebruikt:        HEX_NAAR_BYTE, STRING_NAAR_HOOFDLET, LEES_STRING
; Leest:           TOETSENBORD_INVOER, etc.
; Schrijft:        TOETSENBORD_INVOER, etc.
;-----;
LEES_BYTE PROC NEAR
        PUSH    DX
        MOV     GRENS_AANTAL_TEK,3        ;sta twee tekens toe (plus Enter)
        LEA     DX,TOETSENBORD_INVOER
        CALL    LEES_STRING
        CMP     GELEZEN_AANTAL_TEK,1      ;kijk hoeveel tekens
        JE      ASCII_INVOER              ;maar een, behandel als ASCII-teken
        JB      GEEN_TEKENS                ;alleen Enter-toets ingedrukt
        CALL    STRING_NAAR_HOOFDLET      ;nee, zet string om in hoofdletters
        LEA     DX,TEKENS                 ;adres van om te zetten string
        CALL    HEX_NAAR_BYTE             ;zet string van hex om in byte
        JC      GEEN_TEKENS                ;fout, dus geef 'geen tekens gelezen'
        MOV     AH,1                      ;geef een teken gelezen door
KLAAR_MET_LEZEN:
        POP     DX
        RET
GEEN_TEKENS:

```


Listing 23-2. *vervolg*

```

        XOR     AH,AH                ;zet op 'geen tekens gelezen'
        JMP     KLAAR_MET_LEZEN
ASCII_INVOER:
        MOV     AL,TEKENS            ;laad gelezen tekens
        MOV     AH,1                ;geef een teken gelezen door
        JMP     KLAAR_MET_LEZEN
LEES_BYTE      ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
TOETSENBORD_INVOER LABEL BYTE
GRENS_AANTAL_TEK DB 0                ;lengte van invoerbuffer
GELEZEN_AANTAL_TEK DB 0              ;aantal gelezen tekens
TEKENS         DB 80 DUP (0)         ;een buffer voor toetsenbord-invoer
DATA_SEG      ENDS
END

```

Assembleer Tbd_io opnieuw en link de vier bestanden Test, Tbd_io, Video_io en Cursor om deze versie van LEES_BYTE te kunnen testen.

Op dit punt hebben we twee problemen in verband met LEES_BYTE. Weet u nog van de speciale functietoetsen? Die kunnen we niet lezen met de DOS-functie 0Ah. Dat werkt gewoon niet. Druk maar eens een functietoets in als u Test draait. DOS geeft dan geen twee bytes door, met de eerste op nul gezet, zoals je zou verwachten. We kunnen uitgebreide codes op geen enkele manier lezen met de gebufferde invoer van DOS via de functie 0Ah. We hebben deze functie gebruikt om tekens te kunnen wissen met de backspace-toets voor we op Enter drukken. Maar omdat we geen speciale functietoetsen kunnen lezen, moeten we nu onze eigen LEES_STRING-procedure schrijven. We zullen functie 0Ah moeten vervangen om te zorgen dat we een speciale functietoets kunnen indrukken zonder op Enter te moeten drukken. Het tweede probleem met de DOS-functie 0Ah voor invoer via het toetsenbord houdt verband met het teken voor line feed (nieuwe regel). Druk op Control-Enter (line feed) nadat u een teken hebt ingetikt, en probeer dan backspace. U zult zien dat u dan op de volgende regel staat, en op geen enkele manier naar die erboven terug kunt. Onze nieuwe versie van Tbd_io in het volgende hoofdstuk behandelt het line feed-teken (Control-Enter) als een gewoon teken; bij indrukken van line feed komt de cursor dan niet op de volgende regel te staan.

Maar voor we de problemen met LEES_BYTE en LEES_STRING oplossen, schrijven we een procedure die een decimaal getal zonder teken leest. We zullen deze procedure niet gebruiken in dit boek, maar hij staat wel in de Dskpatch-versie op de schijf die bij dit boek hoort, zodat we Dskpatch bijvoorbeeld kunnen vragen om sectornummer 567 te laten zien.

23.2 Decimale invoer

U weet misschien nog dat het grootste getal zonder teken dat we in een enkel woord kunnen zetten, 65535 is. Als we LEES_STRING gebruiken om een reeks decimale cijfers te lezen, vertellen we DOS niet meer dan zes tekens (vijf cijfers en een carriage

return aan het eind) te lezen. Dat betekent natuurlijk dat LEES_DECIMAAL nog altijd getallen van 65536 tot en met 99999 kan lezen, ook al passen die getallen niet in een woord. We zullen moeten oppassen voor dergelijke getallen en een foutcode doorgeven als LEES_DECIMAAL probeert een getal te lezen dat groter is dan 65535, of als hij probeert een teken te lezen dat niet tussen de nul en negen zit. Om onze string van tot aan vijf cijfers in een woord om te zetten, moeten we vermenigvuldigen zoals we dat in hoofdstuk 1 deden: pak het eerste (meest linkse) cijfer, vermenigvuldig het met tien, plak het tweede cijfer erbij, vermenigvuldig dat met tien, enzovoort. Op deze manier zou je bijvoorbeeld 49856 kunnen schrijven als:

$$4 \cdot 10^4 + 9 \cdot 10^3 + 8 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0$$

of, zoals we de berekening zullen uitvoeren:

$$10 * (10 * (10 * (10 * 4 + 9) + 8) + 5) + 6$$

We moeten natuurlijk wel uitkijken voor fouten bij deze vermenigvuldigingen en terugkeren met de overdrachtvlag gezet wanneer zich een fout voordoet. Hoe weten we nu wanneer we een getal groter dan 65535 proberen te lezen? Bij grotere getallen zal de laatste MUL overlopen in het DX-register. CF (de overdrachtvlag) is aan wanneer DX na een woord-vermenigvuldiging niet nul is, dus we kunnen een JC (*Jump if Carry set*, spring indien overdrachtvlag gezet) gebruiken om een fout af te handelen. Hieronder staat LEES_DECIMAAL, die ook elk cijfer op een fout (wanneer het niet tussen 0 en 9 ligt) controleert. Zet deze procedure in het bestand TBD_IO.ASM:

Listing 23-3. Voeg deze procedure toe aan TBD_IO.ASM

```

PUBLIC LEES_DECIMAAL
-----;
; Deze procedure neemt de uitvoerbuffer van LEES_STRING en zet de
; reeks decimale cijfers om in een woord.
;
; AX      Van decimaal omgezet woord
; CF      Aan als fout, uit als geen fout
;
; Gebruikt: LEES_STRING
; Leest:    TOETSENBORD_INVOER, etc.
; Schrijft: TOETSENBORD_INVOER, etc.
-----;
LEES_DECIMAAL PROC NEAR
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     GRENS_AANTAL_TEK,6      ;maximaal 5 cijfers (65535)
    LEA     DX,TOETSENBORD_INVOER
    CALL    LEES_STRING
    MOV     CL,GELEZEN_AANTAL_TEK   ;vraag aantal gelezen tekens op
    XOR     CH,CH                   ;maak hoge byte van telling 0
    CMP     CL,0                    ;geef fout als geen tekens gelezen
    JLE     ONGELDIGE_DECIMAAL      ;geen tekens gelezen, geef fout
    XOR     AX,AX                   ;begin met getal op 0

```


Listing 23-3. *vervolg*

```

        XOR     BX,BX                ;begin bij begin string
CONVERTEER_CIJFER:
        MOV     DX,10                ;vermenigvuldig getal met 10
        MUL     DX                    ;vermenigvuldig AX met 10
        JC      ONGELDIGE_DECIMAAL    ;CF aan als overloop MUL uit een woord
        MOV     DL,TEKENS[BX]         ;vraag volgende cijfer op
        SUB     DL,'0'                ;en zet om in een nibble (4 bits)
        JS      ONGELDIGE_DECIMAAL    ;onjuist cijfer als < 0
        CMP     DL,9                  ;is dit een onjuist cijfer?
        JA      ONGELDIGE_DECIMAAL    ;ja
        ADD     AX,DX                 ;nee, tel dan op bij getal
        INC     BX                    ;wijs naar volgende teken
        LOOP    CONVERTEER_CIJFER     ;vraag volgende cijfer op
KLAAR_DECIMAAL:
        POP     DX
        POP     CX
        POP     BX
        RET
ONGELDIGE_DECIMAAL:
        STC                          ;zet carry op fout aangeven
        JMP     KLAAR_DECIMAAL
LEES_DECIMAAL ENDP

```

Om zeker te weten dat hij goed werkt, moeten we deze procedure testen met alle grenscondities. Hier is een eenvoudig testprogrammaatje voor LEES_DECIMAAL dat op vrijwel dezelfde manier werkt als dat van LEES_BYTE:

Listing 23-4. Veranderingen TEST.ASM

```

CGROEP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
        ORG     100h

        EXTRN   SCHRIJF_HEX:NEAR, SCHRIJF_DECIMAAL:NEAR
        EXTRN   SCHRIJF_STRING:NEAR, STUUR_CRLF:NEAR
        EXTRN   LEES_DECIMAAL:NEAR

TEST      PROC   NEAR
        LEA     DX,INVOER_PROMPT
        CALL    SCHRIJF_STRING
        CALL    LEES_DECIMAAL
        JC      FOUT
        CALL    STUUR_CRLF
        LEA     DX,GELEZEN_GETAL_PROMPT
        CALL    SCHRIJF_STRING
        MOV     DX,AX
        CALL    SCHRIJF_DECIMAAL
FOUT:     CALL    STUUR_CRLF
        INT     20h
TEST      ENDP

CODE_SEG      ENDS

```

Listing 23-4. *vervolg*

```
DATA_SEG          SEGMENT PUBLIC
INVOER_PROMPT      DB      'Tik decimaal getal : ',0
GELEZEN_GETAL_PROMPT DB      'Gelezen getal: ',0
; en nu dummyvariabelen
      PUBLIC KOPREGEL_NR, DISKDRIVE_NR, KOP_DEEL_1, KOP_DEEL_2
      PUBLIC PROMPTREGEL_NR, HUIDIGE_SECTORNR
KOPREGEL_NR         DB      0
DISKDRIVE_NR        DB      0
KOP_DEEL_1          DB      0
KOP_DEEL_2          DB      0
PROMPTREGEL_NR      DB      0
HUIDIGE_SECTORNR    DB      0
DATA_SEG            ENDS

      END          TEST
```

Weer moeten we vier bestanden linken: Test (als eerste), Tbd__io, Video__io en Cursor. Probeer de grenscondities uit, zowel met geldige als ongeldige cijfers (zoals A, dat geen geldig decimaal cijfer is), en met getallen als 0, 65535 en 65536.

23.3 Samenvatting

We komen later nog op de twee eenvoudig testprocedures terug, wanneer we manieren bespreken waarop u uw eigen programma's kunt schrijven. Dan zullen we ook zien hoe u deze wat verfijndere versie van TEST.ASM kunt gebruiken om een programma te schrijven dat getallen van hex in decimaal omzet.

Maar voorlopig zijn we klaar om door te gaan naar het volgende hoofdstuk, waarin we verbeterde versies van LEES__BYTE en LEES__STRING gaan schrijven.

24 Verbeterde toetsenbord-invoer

24.1 Een nieuwe LEES_STRING 254

DEIN_AANTAL_TK	DB	0
DEINER_AANTAL_TK	DS	0
STRING	DS	80 DUP (0)

De stringbuffer begint bij de tweede byte van de opgegeven buffer, dus de eerste byte van de buffer wordt gebruikt voor de string. De stringbuffer zal dus beginnen met de eerste byte van de buffer, dus de eerste byte van de buffer wordt gebruikt voor de string.

Lijsting 24-1. Voeg deze procedure toe aan het programma.

PROCEDURE	BACKSPACE
EXTRN	SCHRIJF_TK:NEAR

We hadden gezegd dat we de ontwikkeling van Dskpatch op dezelfde manier zouden laten zien als wij het programma hebben geschreven — met inbegrip van fouten en onhandig opgezette procedures, waarvan u er al een paar hebt gezien. In dit hoofdstuk gaan we een nieuwe versie van LEES__BYTE schrijven, die een nogal subtiële fout in Dskpatch veroorzaakt. In het volgende hoofdstuk laten we zien hoe dat foutje kan worden verholpen, maar probeer het eerst zelf op te sporen. (Hint: ga zorgvuldig alle grenscondities voor LEES__BYTE na als Dskpatch erop wordt losgelaten.)

24.1 Een nieuwe LEES__STRING

Onze modulaire aanpak vereist korte procedures, zodat geen enkele procedure moeilijk te begrijpen is. De nieuwe versie van LEES__STRING wordt een voorbeeld van een onhandige procedure: veel te lang. Hij zou in meer verschillende procedures moeten worden gesplitst, maar dat laten we aan u over. Dit boek loopt al ten einde, en we moeten nog enkele procedures doen voor Dskpatch een bruikbaar programma is. We kunnen nu alleen maar de eerste helft van een sector editen, en we kunnen die sector nog niet terugschrijven naar de schijf.

In dit hoofdstuk geven we LEES__STRING een nieuwe procedure, BACKSPACE, die de functie van de backspace-toets in de DOS-functie 0Ah emuleert. Als we de backspace-toets indrukken, zal BACKSPACE het laatst getikte teken wissen, zowel van het scherm als uit de string in het geheugen.

Op het scherm zal BACKSPACE het teken wissen door de cursor een plaats naar links te schuiven, er een spatie overheen te zetten en dan weer een spatie naar rechts te schuiven. Dit geeft hetzelfde resultaat als het wissen met backspace in DOS.

In de buffer zal BACKSPACE een teken wissen door de bufferwijzer, DS:SI+BX, zo te veranderen dat hij naar de een na laagste byte in het geheugen wijst. Met andere woorden, BACKSPACE verlaagt BX gewoon ($BX = BX - 1$). Het teken zal nog in de buffer staan, maar ons programma zal het niet zien. Waarom niet? LEES__STRING vertelt ons hoeveel tekens hij heeft gelezen. Als we proberen meer dan dit aantal uit de buffer te lezen, zien we de tekens die we hebben gewist. Anders zien we ze niet.

We moeten oppassen dat we geen tekens wissen wanneer de buffer leeg is. Denk eraan dat ons gebied met de string-gegevens er ongeveer zo uitziet:

GRENS_AANTAL_TEK	DB	0
GELEZEN_AANTAL_TEK	DB	0
STRING	DB	80 DUP (0)

De stringbuffer begint bij de tweede byte van dit gegevensgebied, ofwel op een offset 2 vanaf het begin. BACKSPACE zal dus geen tekens wissen als BX 2, het begin van de buffer, is, omdat de buffer leeg is als BX gelijk aan 2 is.

Dit is BACKSPACE; zet hem in TBD__IO.ASM:

Listing 24-1. Voeg deze procedure toe aan TBD__IO.ASM

```
PUBLIC BACKSPACE
EXTRN SCHRIJF_TEK:NEAR
```

Listing 24-1. *vervolg*

```

;-----;
; Deze procedure wist tekens, met een tegelijk, uit de buffer en van ;
; het scherm wanneer die buffer niet leeg is. BACKSPACE keert gewoon ;
; terug als de buffer leeg is. ;
; ;
; DS:SI+BX Laatst getikte teken dat nog in buffer staat ;
; ;
; Gebruikt: SCHRIJF_TEK ;
;-----;
BACKSPACE PROC NEAR ;wis een teken
    PUSH AX
    PUSH DX
    CMP BX,2 ;is buffer leeg?
    JE EINDE_BS ;ja, lees volgende teken
    DEC BX ;haal een teken uit buffer
    MOV AH,2 ;haal teken van scherm
    MOV DL,BS
    INT 21h
    MOV DL,20h ;schrijf daar spatie
    CALL SCHRIJF_TEK
    MOV DL,BS ;weer terug
    INT 21h
EINDE_BS:
    POP DX
    POP AX
    RET
BACKSPACE ENDP

```

Door naar de volgende versie van LEES_STRING. Het wordt een hele hap; de listing die u ziet is voor maar één procedure. Het is verreweg de langste procedure die we hebben geschreven, en, zoals gezegd, té lang. Dat komt omdat hij met al die verschillende condities zo ingewikkeld is.

Waarom doet LEES_STRING zo veel? We hebben er nog een paar mogelijkheden aan toegevoegd. Als u de Escape-toets indrukt, zal LEES_STRING de stringbuffer leegmaken en alle tekens van het scherm verwijderen. DOS wist ook alle tekens in de stringbuffer als u Escape indrukt, maar haalt geen tekens van het scherm. Het zet in plaats daarvan een backslash (\) aan het einde van de regel en gaat naar de volgende regel. Onze versie van LEES_STRING wordt veelzijdiger dan de DOS-functie voor het lezen van een string.

LEES_STRING gebruikt drie speciale toetsen: de backspace-toets, Escape en Enter. We zouden de ASCII-codes voor elk van deze toetsen in LEES_STRING kunnen zetten wanneer we ze nodig hebben, maar in plaats daarvan voegen we aan het begin van TBD_IO.ASM enkele definities toe om LEES_STRING leesbaarder te maken. Die definities zien er zo uit:

Listing 24-2. Toevoegingen aan TBD_IO.ASM

```

CGROEP GROUP CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP

BS EQU 8 ;backspace-teken

```


Listing 24-2. *vervolg*

```
CR      EQU      13      ;teken voor carriage return
ESC     EQU      27      ;teken voor Escape

CODE_SEG      SEGMENT PUBLIC
```

Nu komt LEES-STRING. Hoewel hij vrij lang is, ziet u aan de listing wel dat hij niet erg ingewikkeld is — alleen maar lang. Vervang de oude versie van LEES_STRING in TBD_IO.ASM door deze nieuwe versie:

Listing 24-3. De nieuwe LEES_STRING in TBD_IO.ASM

```
      PUBLIC LEES_STRING
      EXTRN  SCHRIJF_TEK:NEAR

;-----;
; Deze procedure verricht een functie die erg lijkt de DOS-functie ;
; OAh. Maar deze functie geeft een speciaal teken door als een functie- ;
; toets of speciale toets wordt ingedrukt - geen carriage return. En ;
; ESC wist de invoer en begint opnieuw. ;
; ;
; DS:DX Adres voor toetsenbord-buffer. De eerste byte moet het ;
; maximum aantal te lezen tekens bevatten (plus een voor ;
; de return) En de tweede byte wordt door deze procedure ;
; gebruikt om het aantal feitelijk gelezen tekens door te ;
; geven. ;
; 0 Geen tekens gelezen ;
; -1 Een speciaal teken gelezen ;
; anders feitelijk gelezen aantal (uitgezonderd ;
; de Enter-toets) ;
; Gebruikt: BACKSPACE, SCHRIJF_TEK ;
;-----;

LEES_STRING PROC NEAR
      PUSH AX
      PUSH BX
      PUSH SI
      MOV SI,DX ;gebruik SI als indexregister en
START_OPNIEUW:
      MOV BX,2 ; BX als offset naar begin van buffer
      MOV AH,7 ;roep invoer zonder controle aan
      INT 21h ; voor CTRL-BREAK en geen echo
      OR AL,AL ;is teken uitgebreide ASCII?
      JZ UITGEBREID ;ja, lees uitgebreide teken
NIET_UTGEBREID:
      CMP AL,CR ;is dit een carriage return?
      JE EINDE_INVOER ;ja, we zijn klaar met invoer
      CMP AL,BS ;is het een backspace-teken?
      JNE NIET_BS ;nee
      CALL BACKSPACE ;ja, verwijder teken
      CMP BL,2 ;is buffer leeg?
      JE START_OPNIEUW ;ja, nu kunnen we uitgebreide ASCII
; weer lezen
      JMP SHORT LEES_VOLGENDE_TEK ;nee, vervolg lezen normale tekens
```

Listing 24-3. *vervolg*

```

NIET_BS:
    CMP     AL,ESC                ;is het een ESC?
    JE      LEDIG_BUFFER          ;ja, maak buffer leeg
    CMP     BL,[SI]               ;controleer of buffer vol is
    JA      BUFFER_VOL            ;buffer is vol
    MOV     [SI+BX],AL            ;bewaar teken anders in buffer
    INC     BX                    ;wijs naar volgende vrije teken in
                                ; buffer

    PUSH    DX
    MOV     DL,AL                 ;stuur teken ook naar scherm
    CALL    SCHRIJF_TEK
    POP     DX

LEES_VOLGENDE_TEK:
    MOV     AH,7
    INT     21h
    OR      AL,AL                 ;een uitgebreid ASCII-teken is ongeldig
                                ; als de buffer niet leeg is
    JNE     NIET_UITGEBREID       ;teken is geldig
    MOV     AH,7
    INT     21h                   ;verwerp uitgebreide teken

;-----;
; Geef een foutconditie aan door een pieptoon- ;
; teken naar het scherm te sturen: chr$(7).    ;
;-----;
FOUT_PIEP:
    PUSH    DX
    MOV     DL,7                  ;geef pieptoon met chr$(7)
    MOV     AH,2
    INT     21h
    POP     DX
    JMP     SHORT LEES_VOLGENDE_TEK ; lees nu volgende teken

;-----;
; Maak de stringbuffer leeg en wis alle tekens ;
; die op het scherm staan afgebeeld.           ;
;-----;
LEDIG_BUFFER:
    PUSH    CX
    MOV     CL,[SI]               ;terugstellen over maximum aantal
    XOR     CH,CH

LEDIG_LUS:
                                ;tekens in buffer. BACKSPACE voorkomt
                                ; dat de cursor te ver terug gaat
    CALL    BACKSPACE
    LOOP    LEDIG_LUS
    POP     CX
    JMP     START_OPNIEUW         ;kan nu uitgebreide ASCII-tekens lezen
                                ; omdat de buffer leeg is

;-----;
; De buffer was vol, dus kan geen teken meer   ;
; lezen. Geef een pieptoon om gebruiker erop   ;
; te attenderen dat buffer vol is.             ;
;-----;
BUFFER_VOL:
    JMP     SHORT FOUT_PIEP       ;als buffer vol, alleen pieptoon geven

```


Listing 24-3. *vervolg*

```

;-----;
; Lees de uitgebreide ASCII-CODE en zet die ;
; als het enige teken in de buffer, geef dan ;
; -1 als het aantal gelezen tekens. ;
;-----;
UITGEBREID: ;lees een uitgebreide ASCII-code
    MOV     AH,7
    INT     21h
    MOV     [SI+2],AL ;zet alleen dit teken in buffer
    MOV     BL,0FFh ;aantal gelezen tekens = -1 voor
                   ; speciaal
    JMP     SHORT EINDE_STRING

;-----;
; Bewaar getelde aantal gelezen tekens en geef ;
; door. ;
;-----;
EINDE_INVOER: ;klaar met invoer
    SUB     BL,2 ;getelde aantal tekens
EINDE_STRING:
    MOV     [SI+1],BL ;geef aantal gelezen tekens door
    POP     SI
    POP     BX
    POP     AX
    RET
LEES_STRING  ENDP

```

Als we de procedure stapsgewijs doornemen, zien we dat LEES_STRING eerst kijkt of we een speciale functietoets hebben ingedrukt. Dat laat hij ons alleen doen als de string leeg is. Als we bijvoorbeeld de F1-toets indrukken na op toets *a* te hebben gedrukt, zal LEES_STRING de F1 toets negeren en een pieptoon geven om ons erop te attenderen dat we op het verkeerde moment een speciale toets hebben ingedrukt. We kunnen echter wel op Escape en daarna F1 drukken omdat de Escape toets tot gevolg heeft dat de LEES_STRING de stringbuffer leegmaakt.

Als LEES_STRING een carriage return-teken leest, zet hij het aantal gelezen tekens in de tweede byte van het string-gebied en keert terug. Onze nieuwe versie van LEES_BYTE kijkt naar deze byte om te zien hoeveel tekens LEES_STRING in feite heeft gelezen.

Vervolgens gaat LEES_STRING na of we een backspace-teken hebben getikt. Als dat het geval is, roept hij BACKSPACE aan om een teken te wissen. Als de stringbuffer dan leeg wordt (BX wordt gelijk aan 2, het begin van de stringbuffer), gaat LEES_STRING terug naar het begin, waar hij een speciale toets kan lezen. Anders leest hij gewoon het volgende teken.

Ten slotte gaat LEES_STRING na of het Escape-teken is ingedrukt. BACKSPACE wist alleen tekens wanneer er tekens in de buffer staan, dus we kunnen de stringbuffer wissen door de BACKSPACE-procedure GRENS_AANTAL_TEK keer aan te roepen, omdat LEES_STRING nooit meer dan GRENS_AANTAL_TEK tekens kan lezen. Elk ander teken wordt in de stringbuffer opgeslagen en naar het scherm gestuurd met SCHRIJF_TEK. Dat wil zeggen, tenzij de buffer vol is.

In het vorige hoofdstuk hebben we LEES_BYTE zo veranderd dat hij geen speciale

functietoetsen kon lezen. We hoeven hier maar een paar regels toe te voegen om te zorgen dat LEES_BYTE kan werken met onze nieuwe versie van LEES_STRING, die wél speciale functietoetsen kan lezen. De volgende veranderingen van LEES_BYTE in TBD_IO.ASM zijn daartoe nodig.

Listing 24-4. Veranderingen LEES_BYTE in TBD_IO.ASM

```

PUBLIC LEES_BYTE
;-----;
; Deze procedure leest een enkel ASCII-teken of een hex-getal
; van twee cijfers.
;
; Zet byte in      AL      Code teken (tenzij AH = 0)
;                  AH      1 als ASCII-teken of hex-getal gelezen
;                  0 als geen tekens gelezen
;                  -1 als een speciale toets gelezen
;
; Gebruikt:        HEX_NAAR_BYTE, STRING_NAAR_HOOFDLET, LEES_STRING
; Leest:           TOETSENBORD_INVOER, etc.
; Schrijft:        TOETSENBORD_INVOER, etc.
;-----;
LEES_BYTE PROC NEAR
    PUSH    DX
    MOV     GRENS_AANTAL_TEK,3      ;sta twee tekens toe (plus Enter)
    LEA     DX,TOETSENBORD_INVOER
    CALL    LEES_STRING
    CMP     GELEZEN_AANTAL_TEK,1    ;kijk hoeveel tekens
    JE      ASCII_INVOER           ;maar een, behandel als ASCII-teken
    JB      GEEN_TEKENS            ;alleen Enter-toets ingedrukt
    CMP     BYTE PTR GELEZEN_AANTAL_TEK, 0FFh ;speciale functietoets?
    JE      SPECIALE_TOETS         ;ja
    CALL    STRING_NAAR_HOOFDLET    ;nee, zet string om in hoofdletters
    LEA     DX,TEKENS              ;adres van om te zetten string
    CALL    HEX_NAAR_BYTE          ;zet string van hex om in byte
    JC      GEEN_TEKENS            ;fout, dus geef 'geen tekens gelezen'
    MOV     AH,1                  ;geef een teken gelezen door
KLAAR_MET_LEZEN:
    POP     DX
    RET
GEEN_TEKENS:
    XOR     AH,AH                  ;zet op 'geen tekens gelezen'
    JMP     KLAAR_MET_LEZEN
ASCII_INVOER:
    MOV     AL,TEKENS              ;laad gelezen teken
    MOV     AH,1                  ;geef een teken gelezen door
    JMP     KLAAR_MET_LEZEN
SPECIALE_TOETS:
    MOV     AL,TEKENS[0]           ;geef scancode door
    MOV     AH,0FFh               ;attendeer met -1 op speciale toets
    JMP     KLAAR_MET_LEZEN
LEES_BYTE ENDP

```

Dskpatch is met de nieuwe versies van LEES_BYTE en LEES_STRING veel prettiger om te gebruiken. Maar er zit een foutje in, zoals we zeiden. Draai Dskpatch

om te proberen dat te vinden en probeer alle grenscondities van LEES_BYTE en HEX_NAAR_BYTE uit.

25 Op zoek naar fouten

25.1 Verbeteren van VERDELER 262

25.2 Samenvatting 263

Als u de nieuwe versie van DSKPATCH met *ag* probeert, wat geen hex-getal is, zult u zien dat Dskpatch niets doet als u de Enter-toets indrukt. Omdat de string *ag* geen hex-getal is, is er niets verkeerd aan dat Dskpatch het negeert, maar het programma moet het op z'n minst van het scherm verwijderen.

Deze fout is van het soort dat we alleen kunnen ontdekken door grondig de grenscondities van een programma te testen. Niet alleen de delen, maar het hele programma. De fout hier is niet te wijten aan LEES_BYTE, ook al kwam hij te voorschijn nadat we die procedure herschreven hadden. Het probleem zit eerder in de manier waarop we VERDELER en EDIT_BYTE hebben geschreven.

EDIT_BYTE is zo ontworpen dat hij SCHRIJF_PROMPTREGEL aanroept om de promptregel te herschrijven en de rest van de regel te wissen. Hierdoor worden alle tekens die we hebben getikt, verwijderd. Maar als we een string als *ag* tikken, meldt LEES_BYTE dat hij een string met lengte 0 heeft gelezen en roept VERDELER EDIT_BYTE niet aan. Wat is nu de oplossing?

25.1 Verbeteren van VERDELER

Er zijn eigenlijk twee manieren om dit probleem op te lossen. De beste oplossing zou zijn om Dskpatch zo te herschrijven dat hij een meer modulaire opbouw krijgt en VERDELER opnieuw op te zetten. Dat doen we niet. Denk eraan: programma's zijn nooit helemaal af, maar ergens moet je ophouden. In plaats daarvan verbeteren we VERDELER zodanig dat hij de promptregel herschrijft wanneer LEES_BYTE een string met lengte nul leest.

Om de fout te herstellen, zijn de volgende wijzigingen in VERDELER (in VERDEEL.ASM) nodig:

Listing 25-1. Veranderingen VERDELER in VERDEEL.ASM

```

PUBLIC VERDELER
EXTRN LEES_BYTE:NEAR, EDIT_BYTE:NEAR
EXTRN SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN EDITOR_PROMPT:BYTE
DATA_SEG ENDS

;-----;
; Dit is de centrale verdeler. Bij normaal wijzigen (editen) en ;
; bekijken, leest deze procedure tekens van het toetsenbord en, als het ;
; teken een opdrachttoets (b.v. een cursortoets) is, roept VERDELER ;
; procedures aan die het eigenlijke werk doen. Dit gebeurt bij speciale ;
; toetsen die vermeld staan in de tabel VERDEEL_TABEL, waarin de ;
; adressen van de procedures vlak na de toetsnamen zijn opgeslagen. ;
; Als het teken geen speciale toets is, moet het rechtstreeks in de ;
; sectorbuffer worden gezet -- dit is de edit-modus. ;
; ;
; Gebruikt: LEES_BYTE, EDIT_BYTE, SCHRIJF_PROMPTREGEL ;
; Leest: EDITOR_PROMPT ;
;-----;

VERDELER PROC NEAR
    PUSH AX
    PUSH BX
    PUSH DX
VERDEEL_LUS:

```

Listing 25-1. *vervolg*

```

CALL    LEES_BYTE           ;lees teken naar AX
OR      AH,AH               ;AX = 0 als geen teken gelezen, -1
                                ; voor code uitgebreide tekenset
JZ      GEEN_TEKENS_GELEZEN ;geen teken gelezen, probeer opnieuw
JS      SPECIALE_TOETS      ;lees uitgebreide code
MOV     DL,AL
CALL    EDIT_BYTE           ;was normaal teken, wijzig byte
JMP     VERDEEL_LUS         ;lees nog een teken

SPECIALE_TOETS:
CMP     AL,68                ;F10--stop?
JE      EINDE_VERDEEL       ;ja, einde
LEA     BX,VERDEEL_TABEL    ;gebruik BX om tabel door te kijken

SPECIALE_LUS:
CMP     BYTE PTR [BX],0      ;einde van tabel?
JE      NIET_IN_TABEL        ;ja, toets stond niet in tabel
CMP     AL,[BX]              ;is het deze tabel-ingang?
JE      VERDEEL              ;ja, dan uitvoeren
ADD     BX,3                 ;nee, probeer volgende ingang
JMP     SPECIALE_LUS         ;controleer volgende tabel-ingang

VERDEEL:
INC     BX                   ;wijs naar adres van procedure
CALL    WORD PTR [BX]        ;roep procedure aan
JMP     VERDEEL_LUS          ;wacht op een andere toets

NIET_IN_TABEL:               ;doe niets, lees gewoon volgende teken
JMP     VERDEEL_LUS

GEEN_TEKENS_GELEZEN:
LEA     DX,EDITOR_PROMPT
CALL    SCHRIJF_PROMPTREGEL ;wis getikte tekens die ongeldig zijn
JMP     VERDEEL_LUS         ;probeer opnieuw

EINDE_VERDEEL:
POP     DX
POP     BX
POP     AX
RET

VERDELER  ** ENDP

```

Deze veroorzaakt geen groot probleem, maar maakt Dskpatch wel iets minder netjes. En een net programma is iets waarnaar we moeten streven. Het is ook vaak duidelijk, en daar zijn onze regels voor het modulair ontwerpen op gericht.

25.2 Samenvatting

VERDELER is een elegante procedure omdat hij een eenvoudige oplossing van een probleem biedt. In plaats van talrijke vergelijkingen voor elk teken dat we zouden kunnen tikken, hebben we een tabel gemaakt die doorzocht kan worden. Dat maakt VERDELER eenvoudiger, en dus betrouwbaarder, dan een programma met allerlei verschillende instructies voor elke conditie die zich zou kunnen voordoen. Met onze

kleine verbetering hebben we VERDELER iets ingewikkelder gemaakt — in dit geval niet veel, maar sommige fouten vergen een veel ingewikkelder procedure.

Als u merkt dat u verbeteringen aan het aanbrengen bent die een procedure te ingewikkeld maken, herschrijf dan de nodige procedures om dat te voorkomen. En check altijd de grenscondities zowel voor- als nadat u een procedure aan uw hoofdprogramma hebt toegevoegd. U bespaart uzelf op die manier een hoop werk bij het debuggen.

We kunnen er niet genoeg de nadruk op leggen hoe belangrijk het is dat u procedures test met grenscondities en u aan de regels van het modulaire ontwerpen houdt. Het leidt allebei tot zowel betere als betrouwbaardere programma's. In het volgende hoofdstuk bekijken we nog een methode voor het debuggen van programma's.

26 Gewijzigde sectoren wegschrijven

- 26.1 Naar de schijf schrijven 266**
- 26.2 Andere methoden van debuggen 268**
- 26.3 Een laadoverzicht opzetten 268**
- 26.4 Fouten opsporen 270**
- 26.5 Symdeb 272**
- 26.6 Samenvatting 274**

We hebben bijna een bruikbaar Dskpatch-programma. In dit hoofdstuk schrijven we een procedure om een gewijzigde sector te kunnen terugschrijven naar de schijf, en in het volgende hoofdstuk een procedure die de tweede helft van een sector toont. Dskpatch is dan nog niet af — zoals we zeiden: een programma is nooit helemaal af — maar we zijn dan wel klaar met wat we in dit boek wilden behandelen. In de versies van Dskpatch op de schijf die bij dit boek verkrijgbaar is, vindt u allerlei extra's.

26.1 Naar de schijf schrijven

Een gewijzigde sector naar een schijf wegschrijven, kan rampzalige gevolgen hebben als het niet bewust gebeurt. Alle Dskpatch-functies zijn tot dusver afhankelijk van de functietoetsen F1, F2 en F10 en de cursortoetsen. Maar deze toetsen kunnen ook stuk voor stuk per ongeluk worden ingedrukt. Gelukkig geldt dat niet voor de functietoetsen met Shift, dus we zullen de F5-toets samen met de Shift-toets gebruiken om een schijfsector te schrijven. Dat geeft de zekerheid dat we alleen een sector naar de schijf terugschrijven als we dat echt willen.

Breng de volgende veranderingen aan in VERDEEL.ASM om SCHRIJF_SECTOR aan de tabel toe te voegen.

Listing 26-1. Veranderingen VERDEEL.ASM

```
DATA_SEG SEGMENT PUBLIC

CODE_SEG SEGMENT PUBLIC
    EXTRN VOLGENDE_SECTOR:NEAR ;in DISK_IO.ASM
    EXTRN VORIGE_SECTOR:NEAR ;in DISK_IO.ASM
    EXTRN FANTOOM_OMHOOG:NEAR, FANTOOM_OMLAAG:NEAR;in FANTOOM.ASM
    EXTRN FANTOOM_LINKS:NEAR, FANTOOM_RECHTS:NEAR
    EXTRN SCHRIJF_SECTOR:NEAR ;in DISK_IO.ASM
CODE_SEG ENDS

;-----;
; Deze tabel bevat de toegestane uitgebreide ASCII-toetsen en de ;
; adressen van de procedures die moeten worden aangeroepen wanneer een ;
; toets wordt ingedrukt. ;
; De tabel heeft de volgende indeling: ;
; DB 72 ;uitgebreide code voor cursor omhoog ;
; DW OFFSET CGROEP:FANTOOM_OMHOOG ;
;-----;

VERDEEL_TABEL LABEL BYTE
    DB 59 ;F1
    DW OFFSET CGROEP:VORIGE_SECTOR
    DB 60 ;F2
    DW OFFSET CGROEP:VOLGENDE_SECTOR
    DB 72 ;cursor omhoog
    DW OFFSET CGROEP:FANTOOM_OMHOOG
    DB 80 ;cursor omlaag
    DW OFFSET CGROEP:FANTOOM_OMLAAG
    DB 75 ;cursor naar links
    DW OFFSET CGROEP:FANTOOM_LINKS
    DB 77 ;cursor naar rechts
    DW OFFSET CGROEP:FANTOOM_RECHTS
```

Listing 26-1. *vervolg*

```

DB      88                                ;Shift-F5
      DW      OFFSET CGROEP: SCHRIJF_SECTOR
      DB      0                                ;einde van de tabel
DATA_SEG ENDS

```

SCHRIJF_SECTOR zelfs is bijna identiek aan LEES_SECTOR. De enige verandering is dat we een sector willen schrijven in plaats van lezen. Terwijl de INT 25h DOS vraagt een sector te lezen, vraagt de tegenovergestelde functie, INT 26H, DOS een sector naar de schijf te schrijven. Dit is SCHRIJF_SECTOR; zet hem in DISK_IO.ASM:

Listing 26-2. Voeg deze sector toe aan DISK_IO.ASM

```

      PUBLIC SCHRIJF_SECTOR
;-----;
; Deze procedure schrijft de sector terug naar de schijf. ;
; ;
; Leest:      DISKDRIVE_NR, HUIDIGE_SECTORNR, SECTOR ;
;-----;
SCHRIJF_SECTOR PROC NEAR
      PUSH    AX
      PUSH    BX
      PUSH    CX
      PUSH    DX
      MOV     AL,DISKDRIVE_NR      ;drive-nummer
      MOV     CX,1                 ;schrijf 1 sector
      MOV     DX,HUIDIGE_SECTORNR ;logische sector
      LEA     BX,SECTOR
      INT     26h                  ;schrijf de sector naar schijf
      POPF    ;verwijder vlag-informatie
      POP     DX
      POP     CX
      POP     BX
      POP     AX
      RET
SCHRIJF_SECTOR ENDP

```

Assembleer nu zowel Dskpatch als Disk_io, maar probeer de schrijffunctie van Dskpatch nog niet uit. Zoek een oude schijf die niet meer belangrijk voor u is en doe hem in drive A. Doe uw programmaschijf dan in een andere drive, bijvoorbeeld B. Draai Dskpatch vanuit drive A (of in welke drive hij ook zit) zodat Dskpatch de eerste sector van uw kladschijf in drive A leest. Kijk voor u verder gaat nog eens goed of het een kladschijf is zodat het niet erg is als er iets misgaat.

Verander een byte in uw sector-afbeelding en noteer welke dat is. Druk dan Shift en F5 in. U ziet dan het ronde lampje van de drive aangaan: u hebt net een gewijzigde sector naar drive A geschreven.

Druk daarna F2 in om de volgende sector (sector 1) te lezen, daarna F1 om de vorige sector (uw oorspronkelijke sector 0) te lezen. U moet de gewijzigde sector dan weer zien. Herstel die oude sector en schrijf hem terug naar drive A om uw kladschijf weer in ere te herstellen.

26.2 Andere methoden van debuggen

Wat zou er gebeuren als we een klein foutje in ons programma hadden gemaakt? Dskpatch is groot genoeg om moeilijkheden te verwachten als we er een fout in willen opsporen. Bovendien bestaat Dskpatch uit negen verschillende bestanden die we moeten linken om tot het bestand DSKPATCH.COM te komen. Hoe vinden we nu een procedure in dit grote programma zonder langzaam het hele programma te moeten doornemen? Zoals u in dit hoofdstuk zult zien, zijn er allerlei manieren om procedures op te zoeken: met een laadoverzicht (*load map*) dat LINK voor ons kan maken, of met het programma SYMDEB van Microsoft, dat we in plaats van DEBUG kunnen gebruiken.

Toen wij bezig waren de oorspronkelijke versie van Dskpatch te schrijven, ging er iets mis toen we SCHRIJF_SECTOR erbij zetten; als we Shift_F5 indrukten, ging de machine op tilt. Maar we konden niet vinden wat er verkeerd was aan SCHRIJF_SECTOR en de enige wijzigingen die we hadden aangebracht, waren die in VERDEEL_TABEL. Alles leek in orde.

Uiteindelijk bleek de fout te zitten in een verkeerde definitie van de Verdeler. De ingang voor SCHRIJF_SECTOR in VERDEEL_TABEL was niet goed. Op de een of andere manier hadden we DW in plaats van DB in de tabel gezet. Het adres van SCHRIJF_SECTOR stond daarom een byte hoger in het geheugen dan waar het moest staan. U ziet de fout hieronder cursief aangegeven:

```
VERDEEL_TABEL LABEL BYTE
                .
                .
                .
                DB 77 ;cursor naar rechts
                DW OFFSET CGROEP:FANTOOM_RECHTS
                DW 88 ;Shift-F5
                DW OFFSET CGROEP:SCHRIJF_SECTOR
                DB 0 ;einde van de tabel
DATA_SEG       ENDS
```

Breng, om het debuggen eens te oefenen, deze wijziging nu eens aan in uw bestand VERDEEL.ASM en volg de aanwijzingen in de volgende paragraaf.

26.3 Een laadoverzicht opzetten

We gaan leren hoe LINK moet worden gebruikt om een overzicht (*map*) van Dskpatch te krijgen. Dat overzicht helpt u bij het opzoeken van procedures en variabelen in het geheugen.

De LINK-opdracht die we tot dusver hebben gebruikt, is vrij lang geworden:

```
LINK DSKPATCH DISK_IO TOON_SEC VIDEO_IO CURSOR VERDEEL TBD_IO FANTOOM EDITOR
```

en er komt nog meer bij. Betekent dat dat we steeds maar bestand na bestand moeten intikken? Nee, er is een veel gemakkelijker manier. LINK kan ons een *automatisch responsbestand* verschaffen dat alle informatie bevat. Met een dergelijk bestand, dat we *linkinfo* zullen noemen, kunnen we gewoon dit tikken:

LINK @LINKINFO

en LINK leest dan zijn informatie uit het bestand.

Met de bestandsnamen die we tot dusver hebben gebruikt, ziet linkinfo er zo uit:

```
DSKPATCH DISK_IO TOON_SEC VIDEO_IO CURSOR +  
VERDEEL TBD_IO FANTOOM EDITOR
```

De plus (+) aan het einde van de eerste regel betekent voor LINK dat hij moet door-
gaan met de bestandsnamen op de volgende regel te lezen.

We kunnen ook informatie toevoegen waardoor LINK een overzicht van de procedu-
res en variabelen in ons programma bij dit eenvoudige linkinfo-bestand maakt. Het
hele linkinfo-bestand ziet er als volgt uit:

```
DSKPATCH DISK_IO TOON_SEC VIDEO_IO CURSOR +  
VERDEEL TBD_IO FANTOOM EDITOR  
±DSKPATCH  
DSKPATCH /MAP;
```

De laatste twee regels zijn nieuwe parameters. De eerste daarvan, *dskpatch* zegt
LINK dat we willen dat het .EXE-bestand DSKPATCH.EXE komt te heten; de twee-
de regel vertelt LINK dat hij een listingbestand genaamd DSKPATCH.MAP moet
aanmaken — ons laadoverzicht. De parameter */map* vertelt LINK dat hij een lijst
moet maken van alle procedures en variabelen die we als *public* hebben gedeclareerd.
Maak het overzichtbestand aan door Dskpatch opnieuw te linken met dit linkinfo-
responsbestand. Het overzichtbestand dat de linker aanmaakt is zo'n 120 regels lang.
Dat is wat te lang om hier in z'n geheel af te beelden, dus we laten u de delen zien
die van bijzonder belang zijn. Dit is de gedeeltelijke listing van het overzichtbestand
DSKPATCH.MAP:

Warning: no stack segment

Start	Stop	Length	Name	Class
00000H	007F5H	007F6H	CODE_SEG	
00800H	0292FH	02130H	DATA_SEG	

Origin	Group
0000:0	CGROEP

Address	Publics by Name
0000:068A	BACKSPACE
0000:06E9	BEWAAR_ECHTE_CURSOR
0000:04C1	CURSOR_RECHTS
0000:0804	DISKDRIVE_NR
0000:081F	EDITOR_PROMPT
0000:07D1	EDIT_BYTE
0000:2922	FANTOOMCURSOR_X
0000:2923	FANTOOMCURSOR_Y
	.
	.
	.
0000:0500	VERDELER

Listing *vervolg*

0000:025F	VOLGENDE_SECTOR
0000:023F	VORIGE_SECTOR
0000:072D	WIS_FANTOOM
0000:049F	WIS_SCHERM
0000:04E1	WIS_TOT_EINDE_REGEL

Address	Publics by Value
---------	------------------

0000:0220	LEES_SECTOR
0000:023F	VORIGE_SECTOR
0000:025F	VOLGENDE_SECTOR
0000:027B	SCHRIJF_SECTOR
0000:02A0	START_SEC_AFB
0000:02FA	TOON_HALVE_SECTOR
.	.
.	.
.	.
0000:0805	REGELS_VOOR_SECTOR
0000:0806	KOPREGEL_NR
0000:0807	KOP_DEEL_1
0000:080D	KOP_DEEL_2
0000:081E	PROMPTREGEL_NR
0000:081F	EDITOR_PROMPT
0000:0854	SECTOR
0000:2922	FANTOOMCURSOR_X
0000:2923	FANTOOMCURSOR_Y

Program entry point at 0000:0100

Dit *laadoverzicht* (zo genoemd omdat het aangeeft waar onze procedure in het geheugen worden geladen) bevat drie hoofdonderdelen. Het eerste bevat de segmenten in ons programma. Dskpatch gebruikt maar twee segmenten, `CODE_SEG` en `DATA_SEG`, die bij elkaar staan, dus die twee segmenten worden genoemd.

Het volgende onderdeel van het laadoverzicht laat de *public* procedures en variabelen zien, in alfabetische volgorde. LINK geeft alleen de procedure en variabelen die u als PUBLIC hebt gedeclareerd — die voor de buitenwereld zichtbaar zijn. Als u een lang programma debugt, kunt u misschien beter alle procedures en variabelen *public* maken om ze in het overzicht te krijgen.

Het laatste deel van het overzicht bevat weer alle procedures en geheugenvariabelen, maar nu in de volgorde waarmee ze in het geheugen staan.

In deze beide opsommingen staan ook de geheugenadressen van elke *public* procedure of variabele. Als u ze goed bekijkt, ziet u dat onze procedure VERDELER op het adres 500h begint. We zullen nu dat adres gebruiken om de fout in Dskpatch op te sporen.

26.4 Fouten opsporen

Als u probeert de versie van Dskpatch met de fout erin te draaien, zult u zien dat alles werkt, behalve Shift-F5, waardoor Dskpatch op onze machine bleef steken. U kunt Shift-F5 maar beter niet proberen — je weet nooit wat er op uw machine ge-

beurt als u het doet.

Omdat behalve Shift-F5 alles werkte (en nu ook werkt), dachten we toen we het programma schreven eerst dat er een fout in SCHRIJF__SECTOR zat. Om die fout op te sporen, zouden we Dskpatch kunnen debuggen door SCHRIJF__SECTOR helemaal te traceren. Maar we pakken het iets anders aan.

We weten dat VERDELER goed werkt, omdat al het andere (de cursortoetsen, F1, F2 en F10) het goed doet. Dat betekent dat VERDELER een goed startpunt is om de fout in Dskpatch te gaan zoeken.

Als u de programmalisting van VERDELER (in hoofdstuk 25) bekijkt, zult u zien dat de instructie

```
CALL    WORD PTR [BX]
```

middenin VERDELER staat omdat hij alle andere routines aanroept. In het bijzonder zal deze CALL-instructie SCHRIJF__SECTOR aanroepen wanneer we Shift-F5 indrukken. Laten we hier met zoeken beginnen.

We gebruiken Debug om Dskpatch te starten met een breakpoint op deze instructie. Dat betekent natuurlijk dat we het adres van deze instructie moeten hebben, en dat kunnen we vinden door VERDELER, die op 500h begint, te disassembleren. Na een U 500h, gevolgd door nog een U-opdracht, moet u de CALL-opdracht te zien krijgen:

```
      .  
      .  
      .  
2C14:0527 EBF2      JMP      051B  
2C14:0529 43        INC      BX  
2C14:052A FF17      CALL     [BX]  
2C14:052C EBD5      JMP      0503  
      .  
      .  
      .
```

Nu we weten dat de CALL-instructie op geheugenplaats 52Ah staat, kunnen we het breakpoint instellen op dit adres, en SCHRIJF__SECTOR dan stap voor stap door-nemen.

Geef allereerst de opdracht G 52A om Dskpatch tot aan deze instructie uit te voeren. U ziet dan Dskpatch beginnen, en wachten tot u een opdracht geeft. Druk Shift-F5 in, omdat dat de opdracht is die problemen geeft. U ziet dan het volgende:

G 052A

```
AX=FF58 BX=28B3 CX=2830 DX=081F SP=FFF6 BP=6DD0 SI=0EE5 DI=0F0B  
DS=2C14 ES=2C14 SS=2C14 CS=2C14 IP=052A  NV UP EI PL NZ NA PO NC  
2C14:052A FF17      CALL     [BX]                      DS:28B3=TB00
```

Het BX-register wijst nu naar een woord dat het adres van SCHRIJF__SECTOR dient te bevatten. Laten we eens kijken of dat het geval is:

Met andere woorden, we proberen een procedure aan te roepen op 7B00h (de lage byte wordt immers het eerst afgebeeld). Maar als we ons geheugenoverzicht bekijken, zien we dat SCHRIJF_SECTOR op 027B moet staan. Sterker nog: we kunnen uit dit overzicht afleiden dat er op 7B00h helemaal *geen* procedure staat. Het adres is volkomen fout.

Toen we zelf bij het opsporen van deze fout zagen dat dit adres niet klopte, hadden we al gauw door waar de kneep zat. We wisten dat VERDELER en de tabellen in wezen in orde waren, omdat alle andere toetsen het goed deden, dus we bekeken de gegevens voor Shift-F5 nog eens nauwkeurig en zagen toen de DW waar DB had moeten staan. Met een laadoverzicht wordt het debuggen veel gemakkelijker. Laten we nu Symdeb eens bekijken.

26.5 Symdeb

Symdeb (*SYMBOLic DEBugger*) is een programma dat Microsoft samen met de macro-assembler in versie 3.00 heeft opgenomen. Zoals u hier zult zien, is Symdeb zo nuttig dat als u het niet hebt, het het overwegen waard is of u niet een nieuwe versie van uw macro-assembler moet aanschaffen.

Omdat zowel Debug als Symdeb door Microsoft zijn geschreven, kent Symdeb de meeste, zo niet alle, opdrachten van Debug. Het bevat ook een aantal opdrachten die u niet in Debug zult aantreffen, plus nog wat mogelijkheden die hun gewicht in goud waard zijn. We zullen twee daarvan in dit hoofdstuk toepassen: symbolisch debuggen en *screen swapping* (schermomwisseling).

26.5.1 Symbolisch debuggen

Symbolisch debuggen, waar Symdeb zijn naam aan te danken heeft, stelt ons in staat procedure- en variablenamen in plaats van adressen bij onze U (disassembleer)-opdrachten te zien te krijgen. Als we bijvoorbeeld Debug gebruiken om de eerste regel van Dskpatch te disassembleren, zien we:

```
2C14:0100-E89C03      CALL    049F
```

Symdeb, echter, geeft te zien:

```
2C14:0100 E89C03      CALL    WIS_SCHERM
```

Welke van de twee is gemakkelijker te lezen? Het lijkt ons wel duidelijk.

26.5.2 Schermomwisseling

De tweede nieuwe opdracht, schermomwisseling, is handig bij het debuggen van Dskpatch. Dskpatch springt over het scherm, en schrijft op verschillende plaatsen. Toen we Debug in de vorige paragraaf gebruikten, begon Debug naar dit scherm te

schrijven en raakten we uiteindelijk het Dskpatch-scherf helemaal kwijt. Symdeb onderhoudt echter twee afzonderlijke schermen: een voor Dskpatch en een voor zichzelf. Als Dskpatch actief is, zien we het Dskpatch-scherf; wanneer Symdeb actief, zien we het Symdeb-scherf. Na de volgende voorbeelden hebt u een beter idee van wat schermomwisseling inhoudt.

Voor we de symbolische debugging-mogelijkheid van Symdeb benutten, moeten we een symbolenbestand met een programma genaamd Mapsym aanmaken. Mapsym maakt van het .MAP (overzicht)-bestand dat we eerder in dit hoofdstuk hebben aangemaakt een symbolenbestand:

```
C>MAPSYM DSKPATCH
```

```
Microsoft (R) Symbol File Generator Version 4.00  
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.
```

```
Program entry point at 0000:0100
```

In dit geval heeft Mapsym een symbolenbestand genaamd DSKPATCH.SYM gecreëerd. Daarna starten we Symdeb met zowel het symbolenbestand als het .COM-bestand:

```
C>SYMDEB /S DSKPATCH.SYM DSKPATCH.COM
```

```
Microsoft (R) Symbolic Debug Utility Version 4.00  
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.
```

```
Processor is [8086]
```

-

De parameter /S in onze opdracht vertelt Symdeb dat het de mogelijkheid tot schermomwisseling moet toepassen. Het gebruikt deze mogelijkheid alleen als ze expliciet wordt opgegeven, omdat schermomwisselen Symdeb aanzienlijk trager kan maken.

Voor we onze vorige debug-sessie herhalen, kijken we eerst even vlug naar het begin van Dskpatch:

-U

```
330E:0100 E89C03      CALL    WIS_SCHERM  
330E:0103 E80403      CALL    SCHRIJF_KOP  
330E:0106 E81701      CALL    LEES_SECTOR  
330E:0109 E89401      CALL    START_SEC_AFB  
330E:010C 8D161F08     LEA     DX,[EDITOR_PROMPT]  
330E:0110 E84003      CALL    SCHRIJF_PROMPTREGEL  
330E:0113 E8EA03      CALL    VERDELER  
330E:0116 CD20        INT     20
```

-

U ziet hoe mooi Symdeb alle namen, in plaats van de adressen, laat zien.

Toen we VERDELER de laatste keer disassembleerden om het adres van CALL WORD PTR [BX] op te zoeken, moesten we eerst het overzichtbestand raadplegen om het adres van de procedure op te zoeken en daarna U 500 tikken om het te disas-

sembleren. Met Symdeb is het leven veel gemakkelijker. We kunnen gewoon U VERDELER tikken om onze procedure te disassembleren.

-U VERDELER

CGROEP:VERDELER:

```
330E:0500 50      PUSH    AX
330E:0501 53      PUSH    BX
330E:0502 52      PUSH    DX
330E:0503 E80401   CALL    LEES_BYTE
330E:0506 0AE4     OR      AH,AH
330E:0508 7426     JZ      VERDELER+30 (0530)
330E:050A 7807     JS      VERDELER+13 (0513)
```

Na nog twee U-opdrachten zien we onze CALL-instructie.

```
330E:0524 83C303   ADD     BX,+03
330E:0527 EBF2     JMP     VERDELER+1B (051B)
330E:0529 43      INC     BX
330E:052A FF17     CALL    [BX]
330E:052C EBD5     JMP     VERDELER+03 (0503)
```

Tik, net als eerst, G 52A, en daarna Shift-F5. Als u Symdeb hebt, ziet u dan Dskpatch de afbeelding op het scherm maken. Daarna keert u na het indrukken van Shift-F5 naar Symdeb terug. Dit keer ziet u echter niet het Dskpatch-schermbild, omdat Symdeb schermen omwisselt. Om het Dskpatch-schermbild weer te krijgen, moet u de backslash (\)-toets en daarna Enter indrukken. Als u het Dskpatch-schermbild ziet, kunt u door op elke andere toets te drukken weer terugkeren naar het scherm van Symdeb. Er is een subtiliteit die u misschien hebt opgemerkt toen we Symdeb gebruikten. Als we de disassembleer-listings bekijken, zien we dit soort instructies:

```
330E:052C EBD5     JMP     VERDELER+03 (0503)
```

in plaats van:

```
330E:052C EBD5     JMP     VERDEEL_LUS
```

Waarom heeft Symdeb de label VERDEEL_LUS niet gebruikt? We hebben de labels in deze procedure niet gedefinieerd als PUBLIC. Als we terug zouden gaan en alle labels in VERDELER declareren als PUBLIC, zouden we deze labels in de disassembleer-listing zien. (Denk eraan om, als u dit doet, het symbolenbestand met Mapsym te 'verversen'.)

26.6 Samenvatting

Hiermee eindigt onze bespreking van de methoden die kunnen worden toegepast bij het opsporen van fouten. We hebben nu nog maar drie hoofdstukken te doen in dit boek. In het volgende hoofdstuk voegen we de procedures toe waarmee de twee halve sectoren over het scherm kunnen worden verplaatst. In de twee laatste hoofdstukken leren we daarna meer over de verschillen tussen .COM- en .EXE-bestanden, en bekij-

ken we voor een laatste keer de ASSUME-opdracht en segment-overrides.
Tussen haakjes: vergeet niet de fout te herstellen die we in VERDEEL__TABEL hebben gezet.

27 De andere halve sector

27.1 Een halve sector verschuiven 278

27.2 Samenvatting 280

In het ideale geval zou Dskpatch zich als een tekstverwerker moeten gedragen wanneer u de cursor onder de onderste regel van de halve sector-afbeelding probeert te verplaatsen. De afbeelding zou een regel naar boven moeten schuiven en onderaan zou een nieuwe regel moeten verschijnen. De versie van Dskpatch op de schijf bij dit boek doet dat ook, maar hier gaan we niet zo verfijnd te werk. In dit hoofdstuk voegen we alleen raamwerk-versies toe van de twee procedures SCHUIF_OMHOOG en SCHUIF_OMLAAG, die de afbeelding verschuiven. In de schijf-versie van Dskpatch kunnen SCHUIF_OMHOOG en SCHUIF_OMLAAG van één tot zestien regels verschuiven (zestien omdat onze halve sector-afbeelding zestien regels bevat). De versie van SCHUIF_OMHOOG en SCHUIF_OMLAAG die we hier aan Dskpatch gaan toevoegen, verschuiven een halve sector, zodat we alleen de eerste of tweede helft van de sector zien.

27.1 Een halve sector verschuiven

Onze oude versies van FANTOOM_OMHOOG en FANTOOM_OMLAAG zetten de cursor terug naar de onder- of bovenkant van de halve sector-afbeelding wanneer we proberen de cursor verder dan de onder- of bovenkant van de afbeelding te verplaatsen. We gaan FANTOOM_OMHOOG en FANTOOM_OMLAAG zodanig wijzigen dat we of SCHUIF_OMHOOG of SCHUIF_OMLAAG aanroepen wanneer de cursor van de boven- of onderkant van het scherm schuift. Deze twee nieuwe procedures zullen de afbeelding verschuiven en de cursor op zijn nieuwe plaats zetten.

De gewijzigde versies van FANTOOM_OMHOOG en FANTOOM_OMLAAG zien er als volgt uit:

Listing 27-1. Veranderingen FANTOOM.ASM

```
FANTOOM_OMHOOG PROC NEAR
    CALL WIS_FANTOOM          ;wis op huidige positie
    DEC FANTOOMCURSOR_Y       ;zet cursor een regel hoger
    JNS STOND_NIET_BOVEN      ;stond niet bovenaan, schrijf cursor
    MOV FANTOOMCURSOR_Y,0     ;stond bovenaan, dus zet daar terug
    CALL SCHUIF_OMLAAG        ;stond bovenaan, schuiven
STOND_NIET_BOVEN:
    CALL SCHRIJF_FANTOOM      ;schrijf fantoom op nieuwe positie
    RET
FANTOOM_OMHOOG ENDP

FANTOOM_OMLAAG PROC NEAR
    CALL WIS_FANTOOM          ;wis op huidige positie
    INC FANTOOMCURSOR_Y       ;zet cursor een regel lager
    CMP FANTOOMCURSOR_Y,16    ;stond hij onderaan?
    JB STOND_NIET_ONDER       ;nee, dus schrijf fantoom
    MOV FANTOOMCURSOR_Y,15    ;stond onderaan, dus zet daar terug
    CALL SCHUIF_OMHOOG        ;stond onderaan, schuiven
STOND_NIET_ONDER:
    CALL SCHRIJF_FANTOOM      ;schrijf de fantoomcursor
    RET
FANTOOM_OMLAAG ENDP
```

Vergeet niet om in de commentaar-kop voor FANTOOM_OMHOOG en FANTOOM_OMLAAG te vermelden dat deze procedures nu SCHUIF_OMHOOG en SCHUIF-OMLAAG gebruiken:

Listing 27-2. Veranderingen FANTOOM.ASM

```

;-----;
; Deze vier procedures verplaatsen de fantoomcursors.
;
; Gebruikt:   WIS_FANTOOM, SCHRIJF_FANTOOM
;             SCHUIF_OMLAAG, SCHUIF_OMHOOG
; Leest:      FANTOOMCURSOR_X, FANTOOMCURSOR_Y
; Schrijft:   FANTOOMCURSOR_X, FANTOOMCURSOR_Y
;-----;

```

SCHUIF_OMHOOG en SCHUIF-OMLAAG zijn beide vrij eenvoudig, omdat ze de afbeelding omwisselen naar de andere halve sector. Als we bijvoorbeeld eerst de eerste halve sector bekijken, en FANTOOM_OMLAAG roept SCHUIF_OMHOOG op, dan zien we de tweede halve sector. SCHUIF_OMHOOG verandert SECTOR_OFFSET in 256, het begin van de tweede halve sector, zet de cursor naar het begin van de sector-afbeelding, schrijft de afbeelding van de tweede halve sector en ten slotte de fantoomcursor bovenin deze afbeelding.

U kunt de details van zowel SCHUIF_OMHOOG als SCHUIF_OMLAAG zien in de volgende listing. Zet ze in FANTOOM.ASM.

Listing 27-3. Voeg deze procedures toe aan FANTOOM.ASM

```

        EXTRN    TOON_HALVE_SECTOR:NEAR, GANAAR_XY:NEAR
DATA_SEG SEGMENT PUBLIC
        EXTRN    SECTOR_OFFSET:WORD
        EXTRN    REGELS_VOOR_SECTOR:BYTE
DATA_SEG ENDS

;-----;
; Deze twee procedures wisselen tussen de twee halve sector-
; afbeeldingen.
;
; Gebruikt:   SCHRIJF_FANTOOM, TOON_HALVE_SECTOR, WIS_FANTOOM,
;             GANAAR_XY, BEWAAR_ECHTE_CURSOR, HERSTEL_ECHTE_CURSOR
; Leest:      REGELS_VOOR_SECTOR
; Schrijft:   SECTOR_OFFSET, FANTOOMCURSOR_Y
;-----;
SCHUIF_OMHOOG PROC    NEAR
        PUSH    DX
        CALL    WIS_FANTOOM           ;verwijder fantoomcursor
        CALL    BEWAAR_ECHTE_CURSOR  ;bewaar positie echte cursor
        XOR     DL,DL                 ;stel cursor in voor halve sector-
                                      ; afbeelding
        MOV     DH,REGELS_VOOR_SECTOR
        ADD     DH,2
        CALL    GANAAR_XY
        MOV     DX,256                ;beeld tweede helft sector af
        MOV     SECTOR_OFFSET,DX
        CALL    TOON_HALVE_SECTOR
        CALL    HERSTEL_ECHTE_CURSOR ;herstel positie echte cursor

```


Listing 27-3. *vervolg*

```

MOV     FANTOOMCURSOR_Y,0      ;cursor bovenaan tweede halve sector
CALL    SCHRIJF_FANTOOM        ;herstel fantoomcursor
POP     DX
RET
SCHUIF_OMHOOG    ENDP

SCHUIF_OMLAAG    PROC    NEAR
PUSH     DX
CALL     WIS_FANTOOM            ;verwijder fantoomcursor
CALL     BEWAAR_ECHTE_CURSOR    ;bewaar positie echte cursor
XOR      DL,DL                  ;stel cursor in voor halve sector-
                                ; afbeelding

MOV      DH,REGELS_VOOR_SECTOR
ADD      DH,2
CALL     GANAAR_XY
XOR      DX,DX                  ;beeld tweede helft sector af
MOV      SECTOR_OFFSET,DX
CALL     TOON_HALVE_SECTOR
CALL     HERSTEL_ECHTE_CURSOR    ;herstel positie echte cursor
MOV      FANTOOMCURSOR_Y,15      ;cursor onderaan twee halve sector
CALL     SCHRIJF_FANTOOM        ;herstel fantoomcursor
POP      DX
RET
SCHUIF_OMLAAG    ENDP

```

SCHUIF_OMHOOG en SCHUIF_OMLAAG werken nu beide goed, hoewel er nog een probleempje mee is in de huidige vorm van Dskpatch. Start Dskpatch en laat de cursor bovenaan het scherm staan. Druk op cursor-omhoog, en u ziet dan dat Dskpatch de eerste halve sector-afbeelding herschrijft. Waarom? We hebben niet op deze grensconditie gecontroleerd. Dskpatch herschrijft het scherm zodra u probeert de cursor tot voorbij de boven- of onderkant van het scherm te verplaatsen. Dit is een mooie uitdaging voor u. Wijzig Dskpatch zodanig dat hij op de twee grenscondities controleert. Als de fantoomcursor bovenin de eerste halve sector-afbeelding staat en u op cursor-omhoog drukt, dient Dskpatch niets te doen. Als u onderin de tweede halve sector-afbeelding staat en op cursor-omlaag drukt, dient Dskpatch ook niets te doen.

27.2 Samenvatting

Dit waren onze laatste werkzaamheden aan Dskpatch in dit boek. Het was onze bedoeling om Dskpatch te laten zien als een 'levend' voorbeeld van de ontwikkeling van een assembleerprogramma, en u tegelijk een bruikbaar programma te bieden met procedures die u in uw eigen programma's kunt gebruiken. Maar de Dskpatch die u hier hebt ontwikkeld, is nog niet zo goed als wel zou kunnen. De schijf-versie van Dskpatch die bij dit boek verkrijgbaar is, bevat nog meer mogelijkheden. En misschien gaat u die schijf-versie zelf nog weer veranderen, want 'een programma is nooit af... maar er komt een tijd dat het naar de gebruikers moet.'

We besluiten dit boek in een ander tempo. In de volgende twee hoofdstukken houden we ons bezig met twee moeilijker onderwerpen: verplaatsen (relocatie) en meer over segmenten.

Deel 4

Losse eindjes

28 Verplaatsen

28.1 Meervoudige segmenten 284

28.2 Verplaatsing 287

28.3 .COM- contra .EXE-programma's 290

Een onderwerp dat altijd in nevelen gehuld lijkt, betreft het verschil tussen .EXE- en .COM-bestanden en de betekenis van verplaatsbare programma's. Laten we in het kader van onze koerswijziging in deze laatste twee hoofdstukken eens kijken naar relocatie en hoe u een programma van meer dan 64K kunt opzetten — niet dat dat per se nodig is, hoewel veel mensen het doen.

28.1 Meervoudige segmenten

Zodra we programma's beginnen te schrijven die meer dan 64K geheugen beslaan, komen we in de problemen met .COM-bestanden. Hoe kan dat? Dat gaan we nu uitleggen.

Allereerst moet elk programma zijn opgebouwd uit een of meer segmenten, elk van ten hoogste 64K. Maar veel programma's gebruiken meer geheugen door meer segmenten te gebruiken: bijvoorbeeld een codesegment voor het programma, een data-segment voor de gegevens en een stapelsegment voor de stapel en tijdelijke gegevens. Als elk van deze segmenten volledig gebruikt werd, zouden we $3 * 64K = 192K$ geheugen gebruiken. Daardoor krijgen we toegang tot meer geheugen, en dat is het punt waarop het verschil tussen .COM- en .EXE-bestanden een rol gaat spelen: .EXE-programma's zijn speciaal voor dit soort klussen bedoeld.

Al onze programma's in dit boek waren .COM-bestanden, met hetzij één segment hetzij één groep. U zult nog weten dat de pseudo-op GROUP niet meer doet dan verscheidene segmenten combineren tot één eenheid die als een segment fungeert. Als we meer dan een enkel segment willen gebruiken om meer dan 64K geheugen te overbruggen, moeten we wat meer werk doen. We geven een voorbeeld.

Ons programma in hoofdstuk 3, dat een reeks tekens afdrukt, kan daar mooi toe dienen. Dat programma zou er, met groepen, in assembleertaal zo uitzien:

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
        ORG     100h
SCHRIJF_STRING PROC    FAR
        MOV     AH,9                ;roep uitvoer string aan
        MOV     DX,OFFSET CGROEP:STRING ;laad adres van string
        INT     21h                ;schrijf string
        INT     20h                ;keer terug naar DOS
SCHRIJF_STRING ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
STRING DB      "Hallo, dit is DOS.$"
DATA_SEG      ENDS

        END     SCHRIJF_STRING
```

De twee segmenten CODE_SEG en DATA_SEG worden in een enkele 64K-groep, CGROEP, gezet en OFFSET CGROEP:STRING geeft de offset van STRING vanaf het begin van de groep CGROEP.

Wanneer DOS een .COM-programma in het geheugen laadt, stelt het alle vier segmentregisters (CS, DS, ES en SS) in op het begin van CGROEP, zodat DS:OFFSET CGROEP:STRING het volledige adres van STRING is. Maar als we nu twee verschillende segmenten en geen groep hadden? Dan hadden we geen limiet van 64K voor twee segmenten: die zou 128K zijn. Hoe zouden we de segmentregisters dan zo instellen dat ze naar hun respectievelijke segmenten wijzen? Door een .EXE-programma te schrijven, waarmee we verscheidene segmenten kunnen gebruiken, die allemaal op een ander adres beginnen.

DOS biedt ons de mogelijkheid segmentregisters voor een .EXE-programma in te stellen met behulp van enkele assembler-instructies. Deze toekenningen zijn niet zo eenvoudig als ze misschien lijken, maar daar komen we zo op. Eerst gaan we SCHRIJF_STRING herschrijven tot een .EXE-programma.

Voor een .EXE-programma moeten we minstens twee segmenten hebben: het code-segment en het stapelsegment. Deze twee segmenten zijn speciale gevallen voor DOS. DOS stelt de vier registers — CS, SS, IP en SP — in wanneer het een .EXE-programma in het geheugen laadt. Het stelt het CS:IP-register in op de eerste instructie waarvan het adres na de pseudo-op END staat. De eerste instructie kan in een .EXE-programma overal staan, terwijl deze instructie in een .COM-programma de eerste instructie in het codesegment *moet* zijn.

Op dezelfde manier wijst SS:IP naar het einde van het stapelsegment dat is gedefinieerd met de pseudo-op SEGMENT STACK. De onderstaande versie van SCHRIJF_STRING bevat bijvoorbeeld een stapel van 80 bytes lang, dus IP zal 80 zijn — het einde van dit stapelgebied binnen het stapelsegment. Dit is het programma:

```
ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STAPEL_SEG
```

```
CODE_SEG      SEGMENT PUBLIC
SCHRIJF_STRING PROC    FAR
    MOV        AX,DATA_SEG      ;segment-adres voor DATA_SEG
    MOV        DS,AX            ;stel DS-register voor DATA_SEG in
    MOV        AH,9             ;roep uitvoer string aan
    MOV        DX,OFFSET STRING ;laad adres van string
    INT        21H              ;schrijf string

    PUSH       ES               ;bewaar terugkeer-adres voor lange
                                ; RET hierna
    XOR        AX,AX            ;er staat een INT 20h op ES:0
    PUSH       AX

    RET                                ;keer terug naar DOS
SCHRIJF_STRING ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
STRING DB      "Hallo, dit is DOS.$"
DATA_SEG      ENDS

STAPEL_SEG    SEGMENT STACK
    DB         10 DUP ('STAPEL ') ;STAPEL gevolgd door twee spaties
```


vervolg

```
STAPEL_SEG      ENDS  
  
                END      SCHRIJF_STRING
```

Dit programma kan worden gedraaid als het hebt gelinkt, maar wis eerst SCHRIJF_STRING.COM. Als u twee versies van een bestand hebt, een met de uitbreiding .COM en een met de uitbreiding .EXE, voert DOS het .COM-bestand uit. Er zijn een aantal verschillen tussen dit .EXE-bestand en ons oorspronkelijke .COM-bestand. In plaats van de INT 20h-instructie om naar DOS terug te keren, hebben we nu allerlei cryptische instructies, te beginnen met PUSH ES. De twee PUSH-instructie zetten een lang adres, ES:0, op de stapel. Dit is het adres van de eerste byte in het 256 bytes lange gegevensgebied dat DOS voor ons programma in het geheugen zet, en de eerste instructie in dit gegevensgebied is een INT 20h-instructie.

Het CS-register moet naar het begin van dit gegevensgebied wijzen wanneer we de INT 20h-instructie uitvoeren. Dit was in ons .COM-programma meteen vanaf het begin het geval. Maar ons .EXE-programma begint met het CS-register ingesteld op het begin van het codesegment, niet van het gegevensgebied. Met een FAR RET naar ES:0 stellen we daarom CS in op het begin van het gegevensgebied en ES:0 bevat, zoals u ziet, de INT 20h-instructie:

```
C>DEBUG SCHRIJFS.EXE  
-U ES:0  
39AF:0000 CD20          INT     20  
39AF:0002 006000        ADD     [BX+SI+00],AH  
.  
.  
.
```

De pseudo-op GROUP ontbreekt omdat we nu drie verschillende segmenten hebben die niet tot één gebied van in totaal hoogstens 64K beperkt zijn. Elk van deze drie segmenten is onafhankelijk, en elk van de segmentregisters (CS, DS en ES) wijst naar een ander segment. Zowel CS als SS worden door DOS ingesteld, zoals we met behulp van Debug kunnen zien:

```
C>DEBUG SCHRIJFS.EXE  
-R  
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000  
DS=39AF  ES=39AF  SS=39C3  CS=39BF  IP=0000  NV UP DI PL NZ NA PO NC  
39BF:0100 B8C139      MOV     AX,39C1
```

DS en ES wijzen naar een lager segment in het geheugen dan CS of SS. Zoals u in hoofdstuk 11 zag, wijzen zowel DS als ES naar het gegevensgebied van 256 bytes dat DOS vóór onze programma's zet. In .COM-programma's reserveerden we dit gebied met de opdracht ORG 100h. Voor .EXE-bestanden hoeven we dat niet te doen omdat het codesegment en het datasegment in verschillende delen van het geheugen staan. Het datasegment staat ergens anders, maar DS wijst niet naar DATA_SEG. Dat is de reden van de eerste instructie in SCHRIJF_STRING. De instructie MOV AX,DATA_SEG verplaatst het segmentnummer van DATA_SEG naar het AX-

register. Als we het programma in het geheugen bekijken:

```
-U
39BF:0000 B8C139      MOV     AX,39C1
39BF:0003 8ED8        MOV     DS,AX
39BF:0005 B409        MOV     AH,09
39BF:0007 BA0000      MOV     DX,0000
39BF:000A CD21        INT     21
39BF:000C 06          PUSH    ES
39BF:000D 33C0        XOR     AX,AX
39BF:000F 50          PUSH    AX
39BF:0010 CB          RETF
39BF:0011 0000        ADD     [BX+SI],AL
```

zien we dat deze MOV-instructie is vertaald in MOV AX,39C1, waarbij 39C1 het segmentnummer voor DATA__SEG is. We hadden twee instructies nodig om dit nummer in het DS-register te zetten, omdat we een getal niet rechtstreeks in een segment-register kunnen zetten. (Zie de adresseermodi in bijlage E-3.)

Waar kwam die 39C1 vandaan? De assembler of de linker wisten vast niet vooraf waar DOS dit programma zou laden; dat weet alleen DOS. Sterker nog: het is DOS dat dit nummer op 39C1 instelt, en het berekenen van dit soort getallen wordt aangeduid met de term relocatie of *verplaatsing*. DOS voert verplaatsingsberekeningen uit voor .EXE-programma's maar niet voor .COM-programma's. Vandaar dat .COM-programma's sneller in het geheugen worden geladen. Ze zijn ook compacter omdat ze niet de speciale informatie bevatten die DOS gebruikt voor het uitvoeren van verplaatsingsberekeningen.

Laten we uit nieuwsgierigheid eens zien wat er gebeurt als we ons .EXE-programma met Exe2bin proberen om te zetten in een .COM-programma:

```
C>EXE2BIN SCHRIJFS.SCHRIJFS.COM
File cannot be converted
C>
```

Exe2bin weet dat het geen .COM-programma van ons bestand kan maken, maar zegt niet waarom niet. Het laat ons dat zelf uitzoeken. Laten we het probleem eens bekijken.

DOS laadt een .COM-programma rechtstreeks in het geheugen nadat het de kop van 256 bytes heeft aangemaakt. Als we, zoals in SCHRIJF__STRING verschillende segmenten willen, en een .COM-bestand willen creëren, moeten we zelf voor de verplaatsing zorgen, met instructies in ons programma. Het is niet erg moeilijk, en we zullen u laten zien hoe het gaat zodat u een beter inzicht krijgt in de manier waarop DOS programma's verplaatst. Als u ooit een groot .COM-programma moet schrijven dat meer dan 64K geheugen nodig heeft, zal deze methode u van pas komen.

28.2 Verplaatsing

Ons doel is om het DS-register in te stellen op het begin van DATA__SEG en het SS-register op het begin van STAPEL__SEGMENT. Dat is te doen met wat handigheden. Eerst moeten we zorgen dat onze drie segmenten in de juiste volgorde in het geheugen worden geladen:

Codesegment
Datasegment
Stapelsegment

Gelukkig hebben we hier al voor gezorgd. De linker laadt deze drie segmenten in het geheugen in de volgorde waarmee ze in ons bestand staan. Wees echter gewaarschuwd: als u de volgende methode in een COM-programma gebruikt om segmentregisters in te stellen, zorg dan dat u de volgorde kent waarin LINK uw segmenten laadt.

Hoe berekenen we de waarde van DS? Laten we eerst eens kijken naar de drie labels die we in de volgende listing in verschillende segmenten hebben gezet. Die labels zijn EINDE_CODE_SEG, EINDE_DATA_SEG en EINDE_STAPEL_SEG. Ze staan niet precies waar je ze zou verwachten. Waarom niet? Omdat, als we een segment definiëren als

```
CODE_SEG      SEGMENT PUBLIC
```

we de linker in feite niet vertellen hoe hij de verschillende segmenten moet combineren. Daarom begint hij elk nieuw segment op een paragraaf-grens — een hex-adres dat met een nul eindigt, zoals 32C40h. Omdat de linker naar de volgende paragraaf-grens springt om met elk nieuw segment te beginnen, zal er vaak een kort, leeg gebied tussen segmenten voorkomen. Door de label EINDE_CODE_SEG aan het begin van DATA_SEG te zetten, gebruiken we ook dit lege gebied. Als we EINDE_CODE_SEG aan het einde van CODE_SEG hadden gezet, zouden we het lege gebied tussen segmenten niet gebruiken. (Bekijk de disassembleer-listing van ons programma maar. U ziet dan een leeg gebied met nullen van 15 bytes lang.) Wat de waarde van het DS-register betreft: DATA_SEG begint op 39AF:0130, of 39C2:0000. De instructie OFFSET CODE_SEG:EINDE_CODE_SEG geeft 130h door, wat het aantal bytes is dat wordt gebruikt door CODE_SEG. Deel dit getal door 16 om het nummer te krijgen dat we nodig hebben om bij DS op te tellen zodat DS naar DATA_SEG wijst. Dezelfde methode gebruiken we om SS in te stellen. De listing van ons programma, met inbegrip van de verplaatsingsinstructies die voor het .COM-bestand nodig zijn, ziet er als volgt uit:

```
      ASSUME  CS:CODE_SEG, DS:DATA_SEG, SS:STAPEL_SEG

CODE_SEG      SEGMENT PUBLIC
      ORG     100h                      ;reserveer gegevensgebied voor
                                           ; .COM-programma
SCHRIJF_STRING PROC    FAR
      MOV     AX,OFFSET CODE_SEG:EINDE_CODE_SEG
      MOV     CL,4                      ;bereken aantal paragrafen
      SHR     AX,CL                     ; (16 bytes) dat codesegment gebruikt
      MOV     BX,CS
      ADD     AX,BX
      MOV     DS,AX                    ;tel CS hierbij op
                                           ;stel DS-register op bij DATA_SEG

      MOV     BX,OFFSET DATA_SEG:EINDE_DATA_SEG
      SHR     BX,CL                     ;bereken door datasegment gebruikte
                                           ; paragrafen
```

```

        ADD     AX,BX                ;tel op bij voor datasegment gebruikte
                                      ; waarde
        MOV     SS,AX                ;stel SS-register in op STAPEL_SEG
        MOV     AX,OFFSET STAPEL_SEG:EINDE_STAPEL_SEG
        MOV     SP,AX                ;stel SP in op einde stapelgebied
        MOV     AH,9                 ;roep uitvoer string aan
        MOV     DX,OFFSET STRING     ;laad adres van string
        INT     21H                  ;schrijf string

        PUSH    ES                   ;bewaar terugkeer-adres voor lange
                                      ; RET hierna
        XOR     AX,AX                ;er staat een INT 20h op ES:0
        PUSH    AX
        RET                           ;keer terug naar DOS
SCHRIJF_STRING ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
EINDE_CODE_SEG LABEL BYTE
STRING DB     "Hallo, dit is DOS.$"
DATA_SEG      ENDS

STAPEL_SEG    SEGMENT PUBLIC
EINDE_DATA_SEG LABEL BYTE
        DB     10 DUP ('STAPEL ') ;STAPEL gevolgd door twee spaties
EINDE_STAPEL_SEG LABEL BYTE
STAPEL_SEG    ENDS

        END     SCHRIJF_STRING

```

Het resultaat van al dit werk kunt u in de volgende Debug-sessie zien:

```

-J
39AF:0100 B83001      MOV     AX,0130
39AF:0103 B104        MOV     CL,04
39AF:0105 D3E8        SHR     AX,CL
39AF:0107 8CCB        MOV     BX,CS
39AF:0109 03C3        ADD     AX,BX
39AF:010B 8ED8        MOV     DS,AX
39AF:010D BB2000      MOV     BX,0020
39AF:0110 D3EB        SHR     BX,CL
39AF:0112 03C3        ADD     AX,BX
39AF:0114 8ED0        MOV     SS,AX
39AF:0116 B85000      MOV     AX,0050
39AF:0119 8BE0        MOV     SP,AX
39AF:011B B409        MOV     AH,09
39AF:011D BA0000      MOV     DX,0000
-U
39AF:0120 CD21        INT     21
39AF:0122 06          PUSH    ES
39AF:0123 33C0        XOR     AX,AX
39AF:0125 50          PUSH    AX
39AF:0126 CB          RETF
39AF:0127 0000        ADD     [BX+SI],AL
39AF:0129 0000        ADD     [BX+SI],AL
39AF:012B 0000        ADD     [BX+SI],AL

```


39AF:012D 0000	ADD	[BX+SI],AL
39AF:012F 004861	ADD	[BX+SI+61],CL
39AF:0132 6C	DB	6C
39AF:0133 6C	DB	6C
39AF:0134 6F	DB	6F
39AF:0135 2C20	SUB	AL,20
39AF:0137 64	DB	64
39AF:0138 69	DB	69
39AF:0139 7420	JZ	015B
39AF:013B 69	DB	69
39AF:013C 7320	JNB	015E
39AF:013E 44	INC	SP
39AF:013F 4F	DEC	DI
-G 120		

AX=0950 BX=0002 CX=0004 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=39C2 ES=39AF SS=39C4 CS=39AF IP=0120 NV UP DI PL NZ NA PO NC
39AF:0120 CD21 INT 21

Door zelf de verplaatsing voor meer dan een segment te laten gelden, hebben we de hoeveelheid geheugen die het .COM-programma kan gebruiken, vergroot. De meeste mensen zullen dit soort trucs nooit nodig hebben, maar door te weten hoe verplaatsing werkt, kunnen we beter begrijpen hoe DOS de verplaatsing van .EXE-bestanden tot stand brengt.

28.3 .COM- contra .EXE-programma's

We besluiten dit hoofdstuk met een samenvatting van het verschil tussen .COM- en .EXE-programma's.

Een op schijf opgeslagen .COM-programma is in wezen een geheugenafbeelding van het programma. Daardoor is een .COM-programma beperkt tot een enkel segment, tenzij het zijn eigen verplaatsing tot stand brengt, zoals we in dit hoofdstuk hebben gedaan.

Een .EXE-programma, daarentegen, laat de verplaatsing door DOS verzorgen. Dat maakt het erg gemakkelijk voor .EXE-programma's om meer segmenten te gebruiken. De meeste grote programma's zijn dan ook .EXE-programma en geen .COM-programma's.

Voor een laatste blik op het verschil tussen .COM- en .EXE-programma's, bekijken we van nabij hoe DOS ze beide laadt en start. Dit maakt het verschil tussen deze twee programmasoorten duidelijker en zichtbaarder. We beginnen met .COM-programma's.

Wanneer DOS een .COM-programma in het geheugen laadt, gaat het als volgt te werk:

- Eerst maakt DOS het programmasegment-prefix (PSP) aan, dat het kladgebied van 256 bytes is dat we in hoofdstuk 11 hebben besproken. Dit PSP bevat onder meer de commandoregel die we tikken.
- Vervolgens kopieert DOS het hele .COM-bestand van de schijf in het geheugen, onmiddellijk na de 256 bytes van het PSP.

- DOS stelt dan alle vier segmentregisters (CS, DS, ES en SS) in op het begin van het PSP.
- Ten slotte stelt DOS het IP-register in op 100h (het begin van het .COM-programma) en het SP-register op het einde van het segment — meestal FFFE, het laatste woord van het segment.

De stappen die nodig zijn bij het laden van een .EXE-bestand zijn hierbij vergeleken wat ingewikkelder, omdat DOS voor de verplaatsing zorgt. Waar vindt DOS de informatie die het nodig heeft voor de verplaatsing?

Zoals blijkt heeft elk .EXE-bestand een kop die is opgeslagen aan het begin van het bestand. Deze kop, of *verplaatsingstabel* is altijd minstens 512 bytes lang en bevat alle informatie die DOS nodig heeft om de verplaatsing tot stand te brengen. Microsoft heeft in de laatste versies van de macro-assembler een programma genaamd EXEMOD opgenomen, dat we kunnen gebruiken om informatie in deze kop te kunnen bekijken:

C>EXEMOD SCHRIJFS

Microsoft (R) EXE File Header Utility Version 4.00
Copyright (C) Microsoft Corp 1985. All rights reserved.

schrijfs	(hex)	(dec)
.EXE size (bytes)	3A0	928
Minimum load size (bytes)	1A0	416
Overlay number	0	0
Initial CS:IP	0000:0000	
Initial SS:SP	0004:0050	0
Minimum allocation (para)	0	0
Maximum allocation (para)	FFFF	65535
Header size (para)	20	32
Relocation table offset	1E	30
Relocation entries	1	1

C>

Onderaan in deze tabel ziet u de verplaatsingsingangen (*Relocation entries*) voor ons programma SCHRIJFS. Steeds als we naar een segmentadres verwijzen, zoals bij MOV AX,DATA__SEG, zal de linker een verplaatsingsingang aan de tabel toevoegen. Het segmentadres is niet bekend voor DOS ons programma in het geheugen laadt, dus we moeten DOS het segmentnummer laten verschaffen.

De tabel bevat nog meer interessante informatie; bijvoorbeeld de beginwaarden van CS:IP en SS:SP. Deze registerparen bevatten de beginwaarden van IP en SP. De tabel vertelt DOS ook hoe veel geheugen ons programma nodig heeft voor het kan draaien: de *Minimum load size*.

Omdat DOS deze verplaatsingstabel gebruikt om de absolute adressen te verschaffen voor verplaatsingen als die van segmentadressen, zijn er nog wat extra stappen nodig wanneer een programma in het geheugen wordt geladen. Dit zijn de stappen die DOS zet bij het laden van een .EXE-programma:

- DOS maakt het programmasegment-prefix (PSP) aan, net als bij een .COM-programma.
- Vervolgens gaat DOS in de .EXE-kop na waar de kop eindigt en het programma begint. Daarna laadt het de rest van het programma in het geheugen achter het PSP.
- Vervolgens zoekt en verbindt DOS met behulp van de informatie in de kop alle verwijzingen in het programma, zoals verwijzingen naar segmentadressen, die moeten worden verplaatst.
- Daarna stelt DOS het ES- en het DS-register zo in dat ze naar het begin van het PSP wijzen. Als uw programma zijn eigen datasegment heeft, moet uw programma DS en/of ES zo wijzigen dat ze naar uw datasegment wijzen.
- Ten slotte stelt DOS het CS-register in op het begin van de codesegment, terwijl IP naar de informatie in de .EXE-kop wijst. Op dezelfde manier stelt het SS:SP in volgens de informatie in de .EXE-kop. In het voorbeeld hierboven geeft de kop aan dat SS:SP op 0004:0050 wordt gezet. Dat betekent dat DOS SP op 0050 zet en SS zo wordt ingesteld dat het vier paragrafen hoger in het geheugen staat dan het einde van het PSP.

29 Meer over segmenten en ASSUME

- 29.1 Segment override 294**
- 29.2 Nog een blik op ASSUME 296**
- 29.3 Phase errors 296**
- 29.4 Slotopmerkingen 297**

In dit laatste hoofdstuk bekijken we het ASSUME-statement nog eens goed en gaan we na welk verband dat houdt met de manier waarop we segmenten gebruiken. In het voorbijgaan leren we over een mogelijkheid genaamd *segment override*, waar we het heel even over hebben gehad. We zullen zien dat segment overrides hand in hand met ASSUME gaan.

29.1 Segment override

Tot dusver hebben we steeds gegevens gelezen en geschreven die in het datasegment stonden. We hebben in dit boek steeds met één segment gewerkt (door gebruik te maken van groepen) zodat we geen reden hadden om gegevens in andere segment te lezen of ernaar te schrijven.

Maar zoals we zagen, bevatten .EXE-programma's meerdere segmenten, en zelfs .COM-programma's kunnen met meer segmenten werken. Een klassiek voorbeeld is het rechtstreeks schrijven naar het scherm. Veel commerciële programma's schrijven naar het scherm door de gegevens naar het schermgeheugen te schrijven en de ROM BIOS-routines volledig te passeren om een hogere snelheid te bereiken. Het schermgeheugen van de IBM-PC voor kleur-grafische adapter staat in segment B800h en voor monochrome video-adapters op B000h. Als we rechtstreeks naar het scherm willen schrijven, moeten we naar andere segmenten schrijven.

In deze paragraaf zullen we een kort programmaatje schrijven dat laat zien hoe we naar twee verschillende segmenten kunnen schrijven met behulp van de registers DS en ES die naar de twee segmenten wijzen. Veel programma's die rechtstreeks naar het schermgeheugen schrijven, gebruiken het ES-register ook om naar het schermgeheugen te wijzen.

Ons programma staat hieronder. Het is erg kort, en u ziet dat het twee datasegmenten heeft, met in elk datasegment een variabele:

```
DATA_SEG      SEGMENT PUBLIC
DS_VAR        DW      1
DATA_SEG      ENDS

EXTRA_SEG     SEGMENT PUBLIC
ES_VAR        DW      2
EXTRA_SEG     ENDS

STAPEL_SEG    SEGMENT STACK
               DB      10 DUP ( 'STAPEL ' )      ;'STAPEL' gevolgd door twee spaties
STAPEL_SEG    ENDS

CODE_SEG      SEGMENT PUBLIC

TEST          ASSUME CS:CODE_SEG, DS:DATA_SEG, ES:EXTRA_SEG, SS:STAPEL_SEG
               PROC     FAR
               PUSH     ES                        ;bewaar terugkeeradres voor lange
               ; RET hierna
               XOR      AX,AX                      ;er staat een INT 20h op ES:0
               PUSH     AX

               MOV      AX,DATA_SEG                ;segmentadres voor DATA_SEG
               MOV      DS,AX                      ;stel DS-register in voor DATA_SEG
```

```

MOV     AX,EXTRA_SEG      ;segmentadres voor EXTRA-SEG
MOV     ES,AX             ;stel ES-register in voor EXTRA_SEG
MOV     AX,DS_VAR         ;lees een variabele uit datasegment
MOV     BX,ES:ES_VAR      ;lees een variabele uit extra segment

RET                                     ;keer terug naar DOS
TEST                                         ENDP
CODE_SEG                                     ENDS
END     TEST

```

We gaan dit programma gebruiken om te zien hoe zowel de pseudo-op ASSUME als segment-overrides werken.

Merk op dat we zowel het datasegment als het stapelsegment *voor* ons codesegment hebben gezet, en dat we ook de pseudo-op ASSUME na alle segment-declaraties hebben gezet. Zoals we in deze paragraaf zullen zien, komt deze volgorde rechtstreeks voort uit het feit dat we twee datasegmenten gebruiken.

Bekijk de twee MOV-instructies in dit programma nog eens goed:

```

MOV     AX,DS_VAR
MOV     BX,ES:ES_VAR

```

De ES: voor de tweede instructie vertelt de 8088 dat hij het ES- in plaats van het DS-register moet gebruiken voor deze bewerking (om de gegevens uit ons extra segment te lezen). Elke instructie gebruikt een standaard segmentregister om naar gegevens te verwijzen. Maar zoals we hier met het ES-register hebben gedaan, kunnen we de 8088 ook vertellen dat we een ander segmentregister voor gegevens willen gebruiken. Dat gaat als volgt. De 8088 heeft vier speciale instructies, één voor elk van de vier segmentregisters: de segment override-instructies. Deze vertellen de 8088 dat hij een specifiek segmentregister in plaats van het standaard register moet gebruiken wanneer de volgende instructie probeert iets uit het geheugen te lezen of ernaar te schrijven.

Onze instructie MOV BX,ES:ES__VAR, bijvoorbeeld, bestaat in feite uit twee instructies. Als u ons testprogramma disassembleert, zult u het volgende zien:

```

2CF4:0011 26          ES:
2CF4:0012 8B1E0000    MOV     BX,[0000]

```

Hieruit blijkt dat de assembler onze instructie heeft vertaald in een segment override-instructie gevolgd door de MOV-instructie. De MOV-instructie zal nu haar gegevens uit het ES- in plaats van het DS-segment lezen.

Als u dit programma traceert, zult u zien dat de eerste MOV-instructie AX gelijk aan 1 maakt (DS__VAR) en de tweede MOV-instructie BX gelijk aan 2 maakt (ES__VAR). Met andere woorden, we hebben gegevens uit twee verschillende segmenten gelezen.

29.2 Nog een blik op ASSUME

Laten we eens kijken wat er gebeurt wanneer we de ES: uit ons programma halen. Verander de regel:

```
MOV    BX,ES:ES_VAR
```

in:

```
MOV    BX,ES_VAR
```

We geven de assembler niet langer opdracht het ES-register te gebruiken wanneer we uit het geheugen lezen, dus hij zal zijn vaste segment (DS) wel weer gebruiken. Maar niet heus.

Gebruik Debug eens om te zien wat deze verandering teweeg heeft gebracht. U zult zien dat de ES:-segment override nog steeds voor onze MOV-instructie staat. Hoe kon de assembler nu weten dat onze variabele in het extra segment en niet in het data-segment staat? Door de informatie te gebruiken die we hem in de pseudo-op ASSUME hebben verschaft.

Ons ASSUME-statement vertelt de assembler dat het DS-register naar het segment DATA__SEG wijst, terwijl ES naar EXTRA__SEG wijst. Bij elke instructie die gebruik maakt van een geheugenvariabele, zoekt de assembler een declaratie van deze variabele om te zien in welk segment die is gedeclareerd. Daarna zoekt hij de ASSUME-lijst door om na te gaan welk segmentregister naar dit segment wijst. De assembler gebruikt dit segmentregister wanneer hij de instructie genereert.

In het geval van onze instructie MOV BX,ES__VAR, zag de assembler dat ES__VAR in het segment genaamd EXTRA__SEG stond, en dat het ES-register naar dat segment wees, zodat hij uit zichzelf een ES: segment override-instructie genereerde. Als we ES__VAR in STAPEL__SEG zouden zetten, zou de assembler een SS: segment override-instructie genereren. De assembler genereert automatisch alle segment override-instructies die we nodig hebben, mits natuurlijk onze pseudo-op ASSUME de feitelijke inhoud van de segmentregisters weerspiegelt.

29.3 Phase errors

Af en toe zult u een geheimzinnige melding van de assembler op het scherm zien, zoals *Phase error between passes*. Deze melding kan een aantal dingen betekenen, maar we richten ons op een bijzonder geval om het u beter te laten begrijpen.

De assembler loopt het programma een aantal keren door bij het genereren van de machinetaal-versie van het programma. Soms verandert daarbij de lengte van het programma, zoals u hier zult zien.

Verplaats in ons voorbeeldprogramma eens de twee datasegmenten (DATA__SEG en EXTRA__SEG) naar een plaats *na* uw codesegment. De assembler zal het hoofdprogramma nu assembleren voor hij zelfs maar naar de datasegmenten kijkt. Als gevolg daarvan zal hij een normale MOV-instructie genereren voor MOV BX,ES__VAR omdat hij niet beseft dat deze variabele in een ander segment staat.

Vervolgens zal de assembler de twee datasegmenten assembleren. Op dit punt aangekomen, zal hij de informatie opslaan dat ES__VAR in het segment EXTRA__SEG

staat. Bij de volgende slag (*pass*) door dit programma, zal de assembler zien dat hij nu ruimte nodig heeft voor een segment override-instructie. Omdat hij de eerste keer geen ruimte voor deze instructie echter heeft gereserveerd, geeft de assembler de melding *Phase error between passes*.

Dat is de reden waarom we al onze datasegmenten voor het codesegment hebben gezet — om de assembler te laten weten welke variabelen in welke segmenten staan. Wat echter niet zo voor de hand ligt, is waarom we het ASSUME-statement in CODE_SEG in plaats van vooraan in het bestand hebben gezet.

We krijgen ook een *phase error*-melding als we onze ASSUME vooraan in het bestand zetten. Om de een of andere reden (die ons niet duidelijk is), moeten we de segmenten *voor* de pseudo-op ASSUME declareren *als* we te maken krijgen met impliciete segment overrides. De veiligste aanpak is daarom om alle gegevens *vóór* het codesegment te declareren en de pseudo-op ASSUME in het codesegment te zetten.

29.4 Slotopmerkingen

U hebt nu veel voorbeelden van programma's in assembleertaal gezien. Dit hele boek door hebben we de nadruk gelegd op het programmeren in plaats van op de details van de 8088-microprocessor binnenin uw IBM Personal Computer. Daarom hebt u niet alle 8088-instructies of pseudo-ops van de assembler gezien. Met wat u hebt geleerd, kunnen de meeste assembleerprogramma's worden geschreven, meer niet. Als u meer wilt weten over het schrijven van assembleerprogramma's, kunt u het beste de programma's in dit boek nemen en er veranderingen in aanbrengen.

Als u een betere manier weet om een bepaald onderdeel van Dskpatch te schrijven, moet u dat zeker doen. Zo hebben wij ook onze eerste programma's leren schrijven. Alle programma's werden toen nog in BASIC geschreven, maar de gedachte geldt nog steeds. We zagen programma's die in BASIC waren geschreven en begonnen over de taal zelf te leren door stukjes van die programma's te herschrijven. Datzelfde kunt u met Dskpatch.

Als u een paar van deze voorbeelden hebt geprobeerd, bent u klaar om uw eigen programma's te schrijven. Begin niet meteen vanuit het niets; dat zou nog te moeilijk zijn. Gebruik om te beginnen de programma's in dit boek als raamwerk. Waag u niet aan een volledig nieuwe structuur of nieuwe methode (uw versie van modulair ontwerpen) voor u helemaal vertrouwd bent met het schrijven van assembleerprogramma's.

Als u echt bezeten raakt van assembleertaal, zult u ook een vollediger boek moeten hebben als naslagboek voor de instructies van de 8088. We geven u hieronder een aantal goede naslagboeken die beschikbaar waren toen wij dit boek schreven. De lijst is geenszins volledig, en de boeken die we noemen zijn niet de enige die we hebben gelezen.

De volgende twee boeken zijn goed geschikt voor programmeurs die iets willen naslaan:

iAPX 88 Book, Intel, 1981. Het absolute naslagboek op dit gebied.

Rector, Russel en Alexy George, *The 8086 Book*, Osborne/McGraw-Hill, 1980. Ook een uitstekend boek, maar nogal moeilijk en ontoegankelijk.

De volgende drie boeken zijn alle drie voor de IBM-PC geschreven. Veel van de informatie in elk ervan is algemeen van aard; alleen de voorbeelden in het laatste deel van deze boeken gelden specifiek voor de IBM-PC. We raden u aan ze alle drie in te zien om na te gaan welke u het interessantst vindt.

Scanlon, Leo, *IBM PC & XT Assembly Language: A Guide for Programmers, Enhanced and Enlarged*, Brady Communication Co., 1985. Dit boek leest erg gemakkelijk. Het vormt een volledige inleiding tot de assembleertaal van de 8088. Als u zich nog wat onzeker voelt over assembleertaal, is dit misschien een goed boek voor u. Zie anders het boek van Morse.

Willen, David C. en Krantz, Jeffrey I., *8088 Assembler Language Programming: The IBM PC*, Howard W. Sams & Co., 1983. Dit is ook een goed boek over de 8088-microprocessor dat is geschreven voor de IBM-PC.

Bradley, David J., *Assembly Language Programming for the IBM Personal Computer*, Prentice-Hall, 1984. De auteur heeft meegewerkt bij het ontwerpen van de IBM-PC en geeft veel voorbeelden voor de IBM-PC. Deze voorbeelden zijn niet volledig, maar kunnen u een idee geven voor programma's die u wilt schrijven. Hij heeft het ook over moeilijkere onderwerpen, zoals de 8087-rekenprocessor, dan de schrijvers van de twee vorige boeken.

Het volgende boek dat we aanbevelen is geen naslagboek en ook geen inleiding tot de IBM-PC. Het is een inleiding tot de 8088-microprocessor, geschreven door een lid van het ontwerpteam van Intel:

Morse, Stephen P., *The 8086/8088 Primer*, Haydn, 1982. Een verrukkelijk boek. Als een van de ontwerpers van Intel vertelt Morse van alles over het ontwerp van de 8088 en over enkele gebreken en fouten die er in het ontwerp van de 8088 zijn geslopen. Hoewel als naslagwerk niet zo geschikt, is het toch volledig en zeer leesbaar en informatief.

Het laatste boek, ten slotte, is als naslagboek nuttig voor iedereen die programmeert op de IBM-PC. We zien het graag als een samenvatting van alles wat een programmeur misschien moet weten over de IBM-PC en de 8086 microprocessor-familie.

Norton, Peter, *Handboek voor IBM-programmeurs*, Kluwer Technische Boeken 1988, ISBN 90 201 2054 9. Omvat een beschrijving van alle DOS- en BIOS-functies, beschrijvingen van belangrijke geheugenplaatsen, een samenvatting van 8086-instructies en een hoop andere nuttige (althans interessante) informatie.

Appendix A

Handleiding bij de diskette

A.1 Voorbeelden uit de hoofdstukken 300

A.2 Uitgebreide versie van Dskpatch 300

Op de diskette bij dit boek staan de meeste Dskpatch-voorbeelden die u in de voorafgaande hoofdstukken hebt gezien, naast een uitgebreide versie van het programma met een groot aantal verbeteringen. De bestanden zijn in twee groepen verdeeld: de voorbeelden uit de hoofdstukken en het verbeterde Dskpatch-programma. In deze appendix wordt uitgelegd wat er op de diskette staat, en waarom.

A.1 Voorbeelden uit de hoofdstukken

Alle voorbeelden komen uit de hoofdstukken 9 tot en 27. De voorbeelden in eerdere hoofdstukken zijn wel zo kort dat u ze snel kunt intikken. Maar vanaf hoofdstuk 9 begonnen we Dskpatch op te bouwen, en aan het einde van het boek was dat uitgegroeid tot negen verschillende bestanden.

In elk hoofdstuk zijn maar enkele van deze negen bestanden veranderd. Omdat ze zich door alle hoofdstukken heen ontwikkelen, was er op de diskette echter niet voldoende ruimte voor elke versie van elk voorbeeld. De voorbeelden op de diskette zien eruit zoals ze er aan het eind van elk hoofdstuk uitzien. Als we dus een programma in, zeg, hoofdstuk 19 meermalen hebben veranderd, bevat de diskette de uiteindelijke versie.

De afbeelding op pagina 304 laat zien wanneer elk bestand verandert. Het laat ook de naam van het schijfbestand voor dat hoofdstuk zien. Als u zeker wilt weten dat u nog op de juiste koers zit, zoek in deze afbeelding dan de naam van de nieuwe bestanden op. Dan kunt u controleren wat u gedaan hebt, of het bestand of de bestanden naar uw schijf kopiëren.

De volledige lijst van alle bestanden op de diskette (met uitzondering van de uitgebreide versie van Dskpatch) ziet er zo uit:

CURSOR14.ASM	CURSOR17.ASM	CURSOR18.ASM	DISK_I15.ASM
DISK_I16.ASM	DISK_I17.ASM	DISK_I19.ASM	DISK_I26.ASM
DSKPAT17.ASM	DSKPAT19.ASM	EDITOR22.ASM	FANTOO21.ASM
FANTOO22.ASM	FANTOO27.ASM	TBD_IO19.ASM	TBD_IO23.ASM
TBD_IO24.ASM	TEST13.ASM	TEST23.ASM	TOON_S14.ASM
TOON_S15.ASM	TOON_S16.ASM	TOON_S17.ASM	TOON_S21.ASM
VERDEE19.ASM	VERDEE22.ASM	VERDEE25.ASM	VERDEE26.ASM
VIDEO_9.ASM	VIDEO_10.ASM	VIDEO_13.ASM	VIDEO_14.ASM
VIDEO_16.ASM	VIDEO_17.ASM	VIDEO_18.ASM	VIDEO_19.ASM
VIDEO_21.ASM			

A.2 Uitgebreide versie van Dskpatch

De diskette bevat meer dan alleen de voorbeelden in dit boek. We hadden Dskpatch aan het eind van hoofdstuk 27 niet echt af, en we hadden nog veel meer in Dskpatch moeten zetten om er een bruikbaar programma van te maken. De diskette bevat een bijna voltooide versie. We geven een kort overzicht van wat er u erop kunt vinden. Zoals het in dit boek staat, kan Dskpatch alleen de volgende of de vorige sector lezen. Als u dus sector 576 zou willen lezen, moet u de F2-toets 575 keer indrukken. Dat is te veel werk. Stel dat u sectoren binnen een bestand wilt lezen. Nu zou u de sector met de directory moeten opzoeken en nagaan waar de sectoren van dat bestand staan. Ook dat is geen lolletje. De diskette-versie van Dskpatch kan hetzij absolute sectoren lezen, net zoals de versie in het boek, of sectoren binnen een

bestand. In de uitgebreide versie is Dskpatch een zeer bruikbaar programma. De uitgebreide versie van Dskpatch bevat te veel wijzigingen om hier in details te beschrijven, dus laten we een paar van de nieuwe functies bekijken die we in de diskette-versie hebben gezet. U kunt veel veranderingen zelf waarnemen als u Dskpatch onderzoekt en uw eigen veranderingen aanbrengt. De uitgebreide versie bestaat eveneens uit negen bestanden, die u allemaal op de diskette aantreft:

DSKPATCH.ASM	VERDEEL .ASM	TOON_SEC.ASM	TBD_IO .ASM	CURSOR .ASM
EDITOR .ASM	FANTOOM .ASM	VIDEO_IO.ASM	DISK_IO .ASM	DSKPATCH.COM

U zult ook een geassembleerde en gelinkte .COM-versie aantreffen die u zonder meer kunt draaien, zodat u de nieuwe versie kunt uitproberen zonder haar te moeten assembleren.

Als u dat doet, zult u allerlei verbeteringen kunnen zien door alleen maar naar de afbeelding op het scherm te kijken. De uitgebreide Dskpatch gebruikt nu acht functietoetsen. Dat is meer dan u kunt onthouden als u Dskpatch niet zo vaak gebruikt, dus de uitgebreide Dskpatch heeft een 'toetsen-regel' onderaan de afbeelding. De functietoetsen werken als volgt.

F1, F2 zijn in dit boek ook gebruikt. F1 leest de vorige sector en F2 de volgende.

- F3 wijzigt het nummer of de letter van de diskdrive. Druk gewoon op F3 en tik een letter, bijvoorbeeld A (zonder dubbele punt), of tik een drivenummer bijvoorbeeld 0. Als u daarna de Enter-toets indrukt, zal Dskpatch naar een andere drive gaan en een sector van de nieuwe diskdrive lezen. U zou Dskpatch zo kunnen veranderen dat hij geen nieuwe sector leest als u van drive verandert. Wij hebben hem zo gemaakt dat het erg moeilijk is om een sector naar de verkeerde schijf te schrijven.
- F4 verandert het sectornummer. Druk gewoon op F4 en tik een sectornummer, in decimaal. Dskpatch leest die sector dan.
- F5 staat in dit boek. Druk Shift en F5 in om een sector terug te schrijven naar de schijf.
- F6 zet Dskpatch in bestandsmodus. Geef gewoon de bestandsnaam en Dskpatch zal een sector uit dat bestand lezen. Van dan af lezen F1 (Vorige sector) en F2 (Volgende sector) sectoren uit dat bestand. F3 beëindigt de bestandsmodus en zet Dskpatch terug in de absolute sector-modus.
- F7 vraagt een offset binnen een bestand. Hij werkt net als F4 (Sector) behalve dat hij sectoren binnen een bestand leest. Als u een offset 3 geeft, leest Dskpatch de vierde sector in uw bestand.
- F10 beëindigt Dskpatch. Als u deze toets per ongeluk indrukt, komt u terug in DOS en verliest u alle veranderingen die u in de laatste sector hebt aange-

bracht. U zou Dskpatch zo kunnen veranderen dat hij eerst vraagt of u echt wilt stoppen met Dskpatch.

Andere veranderingen zijn niet zo duidelijk als die welke u net zag. Dskpatch verschuift het scherm nu bijvoorbeeld met een regel tegelijk. Als u de cursor op de onderste regel van de afbeelding zet en op cursor-omlaag drukt, schuift Dskpatch de afbeelding een regel naar boven en zet er onderaan een nieuwe regel bij. Bovendien werken nu ook enkele andere toetsen van het toetsenbord:

Home zet de fantoomcursor bovenin de halve sector-afbeelding en verschuift de afbeelding zodanig dat u de eerste halve sector ziet.

End zet de fantoomcursor rechts onderin de halve sector-afbeelding en verschuift de afbeelding zodanig dat u de tweede halve sector ziet.

PgUp verschuift de halve sector-afbeelding vier regels. Dit is een aardige mogelijkheid voor als u maar gedeeltelijk door de sector-afbeelding wilt. Als u PgUp snel vier keer indrukt, ziet u de vorige halve sector.

PgDn verschuift de halve sector-afbeelding vier regels in de richting tegenovergesteld aan die van PgUp.

Drive A	Sector 0	
	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	0123456789ABCDEF
00	34 90 49 42 4D 20 20 33 2E 32 00 02 02 01 00	54 IBM 3.2 000
10	02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00	Sp 000 0 0
20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 0F	
30	00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07	
40	BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00	
50	FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1	
60	06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10	
70	7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F	
80	7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03	
90	C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 96	
A0	00 B8 01 02 E8 AA 00 72 19 8B FB B9 0B 00 BE CD	
B0	7D F3 A6 75 0D 8D 7F 20 BE D8 7D B9 0B 00 F3 A6	
C0	74 18 BE 6E 7D E8 61 00 32 E4 CD 16 5E 1F 8F 04	
D0	8F 44 02 CD 19 BE B7 7D EB EB A1 1C 05 33 D2 F7	
E0	36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00	
F0	07 A1 37 7C E8 40 00 A1 18 7C 2A 06 3B 7C 40 50	

Druk op functietoets, of voer teken of hex-byte in:

1Vorig 2Volg. 3Drive 4Sector 5Bew. 6Best. 7Offset 8 9 0Stop

Afb. A-1. De verbeterde versie van Dskpatch.

Desgewenst kunt u de uitgebreide Dskpatch veranderen en aan uw smaak aanpassen. Daarom bevat de diskette ook alle bronbestanden van de uitgebreide Dskpatch. U kunt Dskpatch op die manier wijzigen zoals u wilt en leren van een compleet voorbeeld. U zou bijvoorbeeld de foutcontrole wat kunnen verfraaien. Zoals hij nu is, zet Dskpatch, als u na het indrukken van F2 voorbij het einde van een schijf of bestand raakt, de sector niet terug naar de vorige sector op de schijf of van het bestand. Als u denkt het te kunnen, moet u eens proberen Dskpatch zo te wijzigen dat hij dergelijke fouten opvangt en herstelt.

Of u zou het bijwerken van het scherm kunnen versnellen. Daartoe zou u enkele procedures zoals SCHRIJF__TEK en SCHRIJF__ATTRIBUUT__N__KEER dusdanig moeten veranderen dat ze rechtstreeks naar het schermgeheugen schrijven. Nu gebruiken ze de zeer trage ROM BIOS-routines. Als u echt ambitieus bent, moet u proberen zelf uw tekenuitvoer-routines te schrijven om tekens zeer snel naar het scherm te sturen.

Veel succes.

Hoofd- stuk	DSKPATCH	VERDEEL	TOON_SEC	TBD_IO	CURSOR	EDITOR	FANTOOM	VIDEO_IO	DISK_IO	TEST
9								VIDEO_9 ASM		
10								VIDEO_10 ASM		
13								VIDEO_13 ASM		TEST13 ASM
14			TOON_S14 ASM		CURSOR14 ASM			VIDEO_14 ASM		
15			TOON_S15 ASM						DISK_115 ASM	
16			TOON_S16 ASM					VIDEO_16 ASM	DISK_116 ASM	
17	DSKPAT17 ASM		TOON_S17 ASM		CURSOR17 ASM			VIDEO_17 ASM	DISK_117 ASM	
18					CURSOR18 ASM			VIDEO_18 ASM		
19	DSKPAT19 ASM	VERDEE19 ASM		TBD_IO19 ASM				VIDEO_19 ASM	DISK_119 ASM	
21			TOON_S21 ASM				FANTOO21 ASM	VIDEO_21 ASM		
22		VERDEE22 ASM				EDITOR22 ASM	FANTOO22 ASM			
23				TBD_IO23 ASM						TEST23 ASM
24				TBD_IO24 ASM						
25		VERDEE25 ASM								
26		VERDEE26 ASM							DISK_126 ASM	
27							FANTOO27 ASM			

Appendix B

Listing van Dskpatch

B.1 Beschrijvingen van procedures 306

B.2 Programmalistings van Dskpatch-procedures 313

Deze appendix bevat de definitieve versie van Dskpatch. Als u zelf programma's schrijft, zult u allerlei algemene procedures in deze appendix vinden die u op weg kunnen helpen. We hebben bij elke procedure een korte beschrijving gezet.

B.1 Beschrijvingen van procedures

CURSOR.ASM

WIS__SCHERM

Als het BASIC-commando CLS; wist het tekstschermb.

WIS__TOT__EINDE__REGEL

Wist alle tekens vanaf de cursorpositie tot het einde van de huidige regel.

CURSOR__RECHTS

Zet de cursor een tekenpositie naar rechts zonder een spatie over het oude teken heen te schrijven.

GANAAAR__X__Y

Lijkt veel op het BASIC-commando LOCATE; verplaatst de cursor over het scherm.

STUUR__CRLF

Stuurt een carriage return/line feed naar het scherm. Deze procedure verplaatst de cursor gewoon naar het begin van de volgende regel.

DISK__IO.ASM

VOLGENDE__SECTOR

Telt één bij het huidige sectornummer op, leest die sector dan in het geheugen en herschrijft het Dskpatch-schermb.

VORIGE__SECTOR

Leest de vorige sector. De procedure verlaagt het oude sectornummer (HUIDIGE__SECTORNR) met één en leest de nieuwe sector in de geheugenvariabele SECTOR. Hij herschrijft ook de schermafbeelding.

LEES__SECTOR

Leest een sector (512 bytes) van de schijf in de geheugenbuffer SECTOR.

SCHRIJF__SECTOR

Schrijft een sector (512 bytes) van de geheugenbuffer SECTOR naar de schijf.

VERDEEL.ASM

VERDELER

De centrale verdeelroutine, leest tekens van het toetsenbord en roept dan andere procedures aan die al het werk van Dskpatch doen. Voeg nieuwe opdrachten toe aan VERDEEL__TABEL in dit bestand.

TOON__SEC.ASM

TOON__HALVE__SECTOR

Toont alle hex- en ASCII-tekens die in de halve sector-afbeelding verschijnen door 16 keer TOON__REGEL aan te roepen.

TOON__REGEL

Toont een regel van de halve sector-afbeelding. TOON__HALVE__SECTOR roept deze procedure 16 keer aan om alle 16 regels van de halve sector-afbeelding te laten zien.

START__SEC__AFB

Start de halve sector-afbeelding die u in Dskpatch ziet. Deze procedure herschrijft de halve sector-afbeelding, compleet met kaders en hex-getallen bovenaan, maar zonder kop of prompt.

SCHRIJF__HEX__GETALLEN__BOVENLANGS

Schrijft de regel met hex-getallen bovenlangs de halve sector-afbeelding. Meer kan deze procedure niet.

DSKPATCH.ASM

DISK_PATCH

Het (zeer korte) programmaatje van Dskpatch. DISK_PATCH roept gewoon een aantal andere procedures aan, die al het werk doen. Hij bevat ook veel van de definities voor de variabelen die in Dskpatch worden gebruikt.

EDITOR.ASM

EDIT_BYTE

Bewerkt (*edit*) een byte in de halve sector-afbeelding door een byte zowel in het geheugen (SECTOR) als op het scherm te veranderen. Dskpatch gebruikt deze procedure om bytes in een sector te wijzigen.

SCHRIJF_NAAR_GEHEUGEN

Aangeropen door EDIT_BYTE om een enkele byte in SECTOR te veranderen. Deze procedure verandert de byte die wordt aangewezen met de fantoomcursor.

TBD_IO_ASM

BACKSPACE

Gebruikt door de procedure LEES_STRING om, zodra u op de backspace-toets drukt, een teken te wissen van zowel het scherm als uit de toetsenbordbuffer.

CONVERTEER_HEX_CIJFER

Zet een enkel ASCII-teken om in het overeenkomstige hexadecimale getal. Deze procedure zet bijvoorbeeld de letter A om in het hex-getal 0Ah.

N.B. CONVERTEER_HEX_CIJFER werkt alleen met hoofdletters.

HEX_NAAR_BYTE

Zet een string van twee tekens van een hexadecimale string, zoals A5, om in een enkele byte met die hexwaarde. HEX_NAAR_BYTE verwacht dat de twee tekens cijfers of hoofdletters zijn.

LEES__BYTE

Gebruikt LEES__STRING om een tekenstring te lezen. Deze procedure geeft de speciale functietoets, een enkel teken of een hex-byte door als u een hex-getal van twee cijfers tikt.

LEES__DECIMAAL

Leest een decimaal getal zonder teken van het toetsenbord en gebruikt daarbij LEES__STRING om de tekens te lezen. LEES__DECIMAAL kan getallen van 0 tot en met 65535 lezen.

LEES__STRING

Leest een DOS-achtige tekenstring van het toetsenbord. Deze procedure leest ook speciale functietoetsen, wat de stringleesfunctie van DOS niet doet.

STRING__NAAR__HOOFDLET

Een algemene procedure; zet een DOS-achtige string om in allemaal hoofdletters.

FANTOOM.ASM

WIS__FANTOOM

Haalt de fantoomcursors van het scherm door het tekenattribuut van alle tekens onder de fantoomcursors weer op normaal (7) te zetten.

GA__NAAR__ASCII__POSITIE

Zet de echte cursor naar het begin van de fantoomcursor in het ASCII-venster van de halve sector-afbeelding.

GA__NAAR__HEX__POSITIE

Zet de echte cursor aan het begin van de fantoomcursor in het hex-venster van de halve sector-afbeelding.

FANTOOM__OMLAAG

Schuift de fantoomcursor een regel omlaag in de halve sector-afbeelding, en verschuift het scherm als u probeert lager dan de zestiende regel van de halve sector-afbeelding te gaan.

FANTOOM__LINKS

Schuift de fantoomcursor een ingang naar links, maar niet voorbij de linkerkant van de halve sector-afbeelding.

FANTOOM__RECHTS

Schuift de fantoomcursor een ingang naar rechts, maar niet voorbij de rechterkant van de halve sector-afbeelding.

FANTOOM__OMHOOG

Schuift de fantoomcursor een regel omhoog in de halve sector-afbeelding, of verschuift het scherm als u probeert voorbij de bovenkant van de halve sector-afbeelding te gaan.

HERSTEL__ECHTE__CURSOR

Zet de cursor terug op de plaats die is bewaard door BEWAAR__ECHTE__CURSOR.

BEWAAR__ECHTE__CURSOR

Bewaart de plaats van de echte cursor in twee variabelen. Roep deze procedure aan voor u de echte cursor verplaatst als u hem op zijn plaats wilt terugzetten wanneer u klaar bent met het aanbrengen van veranderingen op het scherm.

SCHUIF__OMLAAG

Toont de eerste helft van de sector. U vindt een uitgebreidere versie van SCHUIF__OMLAAG op de schijf die u bij dit boek kunt krijgen. De uitgebreide versie verschuift de halve sector-afbeelding maar een regel per keer.

SCHUIF__OMHOOG

Aangeropen door FANTOOM__OMLAAG als u probeert de fantoomcursor voorbij de onderkant van de halve sector-afbeelding te schuiven. De versie in dit boek verschuift het scherm niet: ze schrijft de tweede helft van de sector. In de uitgebreide versie op de schijf verschuiven uitgebreide versies van SCHUIF__OMHOOG en SCHUIF__OMLAAG de afbeelding met een regel tegelijk, in plaats van met zestien.

SCHRIJF__FANTOOM

Zet de fantoomcursors in de halve sector-afbeelding: een in het hex-venster en een in het ASCII-venster. Deze procedure verandert gewoon het tekenattribuut in 70H, waardoor zwarte tekens op een witte achtergrond worden afgebeeld.

VIDEO__IO.ASM

Bevat de meeste algemene procedures die u in uw eigen programma's kunt gebruiken.

SCHRIJF__ATTRIBUUT__N__KEER

Een handige procedure die u kunt gebruiken om de attributen van een groep van N tekens te wijzigen. **SCHRIJF__FANTOOM** gebruikt deze procedure om de fantoomcursors te schrijven, en **WIS__FANTOOM** gebruikt hem om de fantoomcursors te verwijderen.

SCHRIJF__TEK

Schrijft een teken naar het scherm. Omdat hij de ROM BIOS-routines gebruikt, hecht deze procedure geen bijzondere betekenis aan een bepaald teken. Een carriage return-teken verschijnt op het scherm als een muzieknoot (het teken voor 0DH). Roep **STUUR__CRLF** aan als u de cursor aan het begin van de volgende regel wilt hebben.

SCHRIJF__TEK__N__KEER

Schrijft N kopieën van een teken naar het scherm. Deze procedure is handig bij het schrijven van lijnen naar het scherm, zoals die welke in patronen worden gebruikt.

SCHRIJF__DECIMAAL

Schrijft een woord naar het scherm in de vorm van een decimaal getal zonder teken in het bereik 0 tot 65535.

SCHRIJF__KOP

Schrijft de kop aan de bovenkant van de afbeelding die u in **Dskpatch** ziet. De procedure toont daar de letter van de diskdrive en het nummer van de sector die u in de halve sector-afbeelding ziet.

SCHRIJF__HEX

Schrijft een getal van een byte naar het scherm als een hex-getal van twee cijfers.

SCHRIJF__HEX__CIJFER

Schrijft een hex-getal van een cijfer op het scherm. Deze procedure zet een nibble van vier bits om in het ASCII-teken en schrijft het naar het scherm.

SCHRIJF__PATROON

Tekent kaders rond de halve sector-afbeelding zoals die door een patroon is gedefinieerd. Gebruik SCHRIJF__PATROON om willekeurige patronen met tekens op het scherm te tekenen.

SCHRIJF__STRING

Een uiterst nuttige, algemene procedure waarmee u een tekenstring naar het scherm kunt schrijven. Het laatste teken van uw string moet een nul-byte zijn.

SCHRIJF__PROMPTREGEL

Schrijft een string op de prompt-regel, wist dan de rest van de regel om tekens van de oude prompt te verwijderen.

B.2 Programmalistings van Dskpatch-procedures

B.2.1 Make-bestand DSKPATCH

Dit is het Make-bestand dat u bij het Make-programma van Microsoft kunt gebruiken om Dskpatch automatisch aan te maken:

```
DSKPATCH.OBJ:  DSKPATCH.ASM
               MASM DSKPATCH;

DISK_IO.OBJ:   DISK_IO.ASM
               MASM DISK_IO;

TOON_SEC.OBJ:  TOON_SEC.ASM
               MASM TOON_SEC;

VIDEO_IO.OBJ:  VIDEO_IO.ASM
               MASM VIDEO_IO;

CURSOR.OBJ:    CURSOR.ASM
               MASM CURSOR;

VERDEEL.OBJ:   VERDEEL.ASM
               MASM VERDEEL;

TBD_IO.OBJ:    TBD_IO.ASM
               MASM TBD_IO;

FANTOOM.OBJ:   FANTOOM.ASM
               MASM FANTOOM;

EDITOR.OBJ:    EDITOR.ASM
               MASM EDITOR;

DSKPATCH.COM:  DSKPATCH.OBJ DISK_IO.OBJ TOON_SEC.OBJ VIDEO_IO.OBJ CURSOR.OBJ \
               VERDEEL.OBJ TBD_IO.OBJ FANTOOM.OBJ EDITOR.OBJ
               LINK @LINKINFO
               EXE2BIN DSKPATCH DSKPATCH.COM
```


B.2.2 CURSOR.ASM

```

CR      EQU      13                      ;carriage return
LF      EQU      10                      ;line feed

CGROEP  GROUP    CODE_SEG
        ASSUME   CS:CGROEP

CODE_SEG      SEGMENT PUBLIC

                PUBLIC  STUUR_CRLF
;-----;
; Deze routine stuurt alleen een carriage return en line feed naar het ;
; scherm met gebruik van de DOS-routines zodat het verschuiven van de ;
; cursor op de juiste wijze wordt verwerkt. ;
;-----;
STUUR_CRLF    PROC NEAR
                PUSH    AX
                PUSH    DX
                MOV     AH,2
                MOV     DL,CR
                INT     21h
                MOV     DL,LF
                INT     21h
                POP     DX
                POP     AX
                RET
STUUR_CRLF    ENDP

                PUBLIC  WIS_SCHERM
;-----;
; Deze procedure wist het hele scherm. ;
;-----;
WIS_SCHERM    PROC      NEAR
                PUSH    AX
                PUSH    BX
                PUSH    CX
                PUSH    DX
                XOR     AL,AL                ;maak hele venster leeg
                XOR     CX,CX                ;linker bovenhoek is (0,0)
                MOV     DH,24                ;onderste regel van scherm is regel 24
                MOV     DL,79                ;rechterkant is kolom 79
                MOV     BH,7                 ;normale attribuut voor lege regels
                MOV     AH,6                 ;roep functie voor omhoog schuiven aan
                INT     10h                  ;maak venster leeg
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                RET
WIS_SCHERM    ENDP

                PUBLIC  GANAAR_XY

```

CURSOR.ASM *vervolg*

```

;-----;
; Deze procedure verplaatst de cursor.
;-----;
;      DH      Rij (Y)
;      DL      Kolom (X)
;-----;
GANAAR_XY      PROC      NEAR
    PUSH      AX
    PUSH      BX
    MOV       BH,0          ;schermpagina 0
    MOV       AH,2          ;aanroep Cursorpositie instellen
    INT       10h
    POP       BX
    POP       AX
    RET
GANAAR_XY      ENDP

PUBLIC  CURSOR_RECHTS
;-----;
; Deze procedure zet de cursor een plaats naar rechts of op de volgende
; regel als de cursor aan het einde van een regel stond.
;-----;
; Gebruikt:      STUUR_CRLF
;-----;
CURSOR_RECHTS  PROC      NEAR
    PUSH      AX
    PUSH      BX
    PUSH      CX
    PUSH      DX
    MOV       AH,3          ;lees huidige cursorpositie
    MOV       BH,0          ;op pagina 0
    INT       10h          ;lees cursorpositie
    MOV       AH,2          ;stel nieuwe cursorpositie in
    INC       DL            ;zet kolom op volgende positie
    CMP       DL,79         ;zorg dat kolom <= 79
    JBE       OK
    CALL      STUUR_CRLF    ;ga naar volgende regel
    JMP       KLAAR
OK:           INT       10h
KLAAR:        POP       DX
              POP       CX
              POP       BX
              POP       AX
              RET
CURSOR_RECHTS  ENDP

PUBLIC  WIS_TOT_EINDE_REGEL
;-----;
; Deze procedure wist de regel vanaf de huidige cursorpositie tot het
; einde van die regel.
;-----;
WIS_TOT_EINDE_REGEL  PROC      NEAR
    PUSH      AX
    PUSH      BX
    PUSH      CX

```


CURSOR.ASM *vervolg*

```

        PUSH    DX
        MOV     AH,3                ;lees huidige cursorpositie
        XOR     BH,BH              ; op pagina nul
        INT     10h                ;(X,Y) nu in DL, DH
        MOV     AH,6                ;klaarmaken voor wissen tot einde regel
        XOR     AL,AL              ;maak venster leeg
        MOV     CH,DH              ;allemaal op dezelfde regel
        MOV     CL,DL              ;begin op plaats van de cursor
        MOV     DL,79              ;en stop bij einde van de regel
        MOV     BH,7                ;gebruik normaal attribuut
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
WIS_TOT_EINDE_REGEL    ENDP

CODE_SEG    ENDS

        END

```

B.2.3 DISK_IO.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG  SEGMENT PUBLIC
        PUBLIC  LEES_SECTOR
DATA_SEG  SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE
        EXTRN   DISKDRIVE_NR:BYTE
        EXTRN   HUIDIGE_SECTORNR:WORD
DATA_SEG  ENDS

        EXTRN   START_SEC_AFB:NEAR

;-----;
; Deze procedure leest de eerste sector op schijf A en dumpt de eerste ;
; helft van deze sector. ;
;-----;
LEES_SECTOR  PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,DISKDRIVE_NR      ;drive-nummer
        MOV     CX,1                 ;lees maar 1 sector
        MOV     DX,HUIDIGE_SECTORNR ;logische sector-nummer
        LEA     BX,SECTOR             ;waar deze sector moet worden
                                         ; opgeslagen
        INT     25h                  ;lees de sector
        POPF    ;verwijder door DOS op stapel
                                         ; gezette vlaggen
        XOR     DX,DX                 ;maak offset binnen SECTOR gelijk aan 0
        CALL    START_SEC_AFB         ;dump eerste helft
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LEES_SECTOR  ENDP

        PUBLIC  VORIGE_SECTOR
        EXTRN   START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
        EXTRN   SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG  SEGMENT PUBLIC
        EXTRN   HUIDIGE_SECTORNR:WORD, EDITOR_PROMPT:BYTE
DATA_SEG  ENDS

;-----;
; Deze procedure leest, zo mogelijk, de vorige sector. ;
; ;
; Gebruikt:  SCHRIJF_KOP, LEES_SECTOR, START_SEC_AFB ;
;           SCHRIJF_PROMPTREGEL ;
; Leest:    HUIDIGE_SECTORNR, EDITOR_PROMPT ;
; Schrijft:  HUIDIGE_SECTORNR ;
;-----;
VORIGE_SECTOR  PROC NEAR
        PUSH    AX
        PUSH    DX

```


DISK_IO.ASM *vervolg*

```

MOV     AX,HUIDIGE_SECTORNR    ;vraag huidige sectornummer op
OR      AX,AX                  ;niet verlagen als reeds 0
JZ      SECTOR_NIET_VERLAGEN
DEC     AX
MOV     HUIDIGE_SECTORNR,AX    ;bewaar nieuwe sectornummer
CALL    SCHRIJF_KOP
CALL    LEES_SECTOR
CALL    START_SEC_AFB          ;beeld nieuwe sector af
LEA     DX,EDITOR_PROMPT
CALL    SCHRIJF_PROMPTREGEL
SECTOR_NIET_VERLAGEN:
POP     DX
POP     AX
RET
VORIGE_SECTOR      ENDP
PUBLIC  VOLGENDE_SECTOR
EXTRN   START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
EXTRN   SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG      SEGMENT PUBLIC
EXTRN   HUIDIGE_SECTORNR:WORD, EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
;-----;
; Leest de volgende sector. ;
; ;
; Gebruikt:  SCHRIJF_KOP, LEES_SECTOR, START_SEC_AFB ;
;           SCHRIJF_PROMPTREGEL ;
; Leest:    HUIDIGE_SECTORNR, EDITOR_PROMPT ;
; Schrijft: HUIDIGE_SECTORNR ;
;-----;
VOLGENDE_SECTOR PROC      NEAR
PUSH     AX
PUSH     DX
MOV      AX,HUIDIGE_SECTORNR
INC      AX                    ;ga naar volgende sector
MOV      HUIDIGE_SECTORNR,AX
CALL     SCHRIJF_KOP
CALL     LEES_SECTOR
CALL     START_SEC_AFB        ;beeld nieuwe sector af
LEA      DX,EDITOR_PROMPT
CALL     SCHRIJF_PROMPTREGEL
POP      DX
POP      AX
RET
VOLGENDE_SECTOR ENDP

PUBLIC  SCHRIJF_SECTOR
;-----;
; Deze procedure schrijft de sector terug naar de schijf. ;
; ;
; Leest:    DISKDRIVE_NR, HUIDIGE_SECTORNR, SECTOR ;
;-----;
SCHRIJF_SECTOR PROC      NEAR
PUSH     AX
PUSH     BX
PUSH     CX

```

DISK_IO.ASM *vervolg*

```
    PUSH    DX
    MOV     AL,DISKDRIVE_NR      ;drive-nummer
    MOV     CX,1                 ;schrijf 1 sector
    MOV     DX,HUIDIGE_SECTORNR ;logische sector
    LEA     BX,SECTOR
    INT     26h                 ;schrijf de sector naar schijf
    POPF    ;verwijder vlag-informatie
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
SCHRIJF_SECTOR ENDP

CODE_SEG ENDS

DATA_SEG SEGMENT PUBLIC
    EXTRN  SECTOR:BYTE
DATA_SEG ENDS

END
```


B.2.4 DSKPATCH.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC
               ORG      100h

               EXTRN    WIS_SCHERM:NEAR, LEES_SECTOR:NEAR
               EXTRN    START_SEC_AFB:NEAR, SCHRIJF_KOP:NEAR
               EXTRN    SCHRIJF_PROMPTREGEL:NEAR, VERDELER:NEAR
DISK_PATCH    PROC NEAR
               CALL     WIS_SCHERM
               CALL     SCHRIJF_KOP
               CALL     LEES_SECTOR
               CALL     START_SEC_AFB
               LEA      DX,EDITOR_PROMPT
               CALL     SCHRIJF_PROMPTREGEL
               CALL     VERDELER
               INT      20h
DISK_PATCH    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC

               PUBLIC   SECTOR_OFFSET
;-----;
; SECTOR_OFFSET is de offset van de halve ;
; sector-afbeelding in de volledige sector. Hij ;
; moet een meervoud van 16 zijn, en niet groter ;
; dan 256. ;
;-----;
SECTOR_OFFSET DW      0

               PUBLIC   HUIDIGE_SECTORNR, DISKDRIVE_NR
HUIDIGE_SECTORNR DW      0          ;aanvankelijk sector 0
DISKDRIVE_NR      DB      0          ;aanvankelijk drive A:

               PUBLIC   REGELS_VOOR_SECTOR, KOPREGEL_NR
               PUBLIC   KOP_DEEL_1, KOP_DEEL_2
;-----;
; REGELS_VOOR_SECTOR is het aantal regels ;
; bovenaan op het scherm voor de halve sector- ;
; afbeelding. ;
;-----;
REGELS_VOOR_SECTOR DB      2
KOPREGEL_NR        DB      0
KOP_DEEL_1         DB      'Drive ',0
KOP_DEEL_2         DB      '      Sector ',0
               PUBLIC   PROMPTREGEL_NR, EDITOR_PROMPT
PROMPTREGEL_NR     DB      21
EDITOR_PROMPT      DB      'Druk op functietoets, of voer'
                  DB      'teken of hex-byte in: ',0

               PUBLIC   SECTOR

```

DSKPATCH.ASM *vervolg*

```
;-----;
; De hele sector (tot aan 8192 bytes) is      ;
; opgeslagen in dit deel van het geheugen.    ;
;-----;
SECTOR DB      8192 DUP (0)

DATA_SEG      ENDS

      END      DISK_PATCH
```


B.2.5 EDITOR.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

CODE_SEG      SEGMENT PUBLIC

DATA_SEG      SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   FANTOOMCURSOR_X:BYTE
        EXTRN   FANTOOMCURSOR_Y:BYTE
DATA_SEG      ENDS

;-----;
; Deze procedure schrijft een byte naar SECTOR, op de geheugenplaats ;
; die door de fantoomcursors wordt aangewezen. ;
; ;
;      DL      Naar SECTOR te schrijven byte ;
; ;
; De offset wordt berekend met ;
;   OFFSET = SECTOR_OFFSET + (16 * FANTOOMCURSOR_Y) + FANTOOMCURSOR_X ;
; ;
; Leest:      FANTOOMCURSOR_X, FANTOOMCURSOR_Y, SECTOR_OFFSET ;
; Schrijft:   SECTOR ;
;-----;
SCHRIJF_NAAR_GEHEUGEN  PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,SECTOR_OFFSET
        MOV     AL,FANTOOMCURSOR_Y
        XOR     AH,AH
        MOV     CL,4 ;vermenigvuldig FANTOOMCURSOR_Y met 16
        SHL     AX,CL
        ADD     BX,AX ;BX = SECTOR_OFFSET + (16 * Y)
        MOV     AL,FANTOOMCURSOR_X
        XOR     AH,AH
        ADD     BX,AX ;dat is het adres!
        MOV     SECTOR[BX],DL ;sla nu de byte op
        POP     CX
        POP     BX
        POP     AX
        RET
SCHRIJF_NAAR_GEHEUGEN      ENDP

        PUBLIC  EDIT_BYTE
        EXTRN   BEWAAR_ECHTE_CURSOR:NEAR, HERSTEL_ECHTE_CURSOR:NEAR
        EXTRN   GA_NAAR_HEX_POSITIE:NEAR, GA_NAAR_ASCII_POSITIE:NEAR
        EXTRN   SCHRIJF_FANTOOM:NEAR, SCHRIJF_PROMPTREGEL:NEAR
        EXTRN   CURSOR_RECHTS:NEAR, SCHRIJF_HEX:NEAR, SCHRIJF_TEK:NEAR
DATA_SEG      SEGMENT PUBLIC
        EXTRN   EDITOR_PROMPT:BYTE
DATA_SEG      ENDS

```

EDITOR.ASM *vervolg*

```

;-----;
; Deze procedure verandert een byte in het geheugen en op het scherm. ;
; ;
; DL      Byte die in SECTOR moet worden geschreven en op scherm ;
;          veranderd ;
; ;
; Gebruikt: BEWAAR_ECHTE_CURSOR, HERSTEL_ECHTE_CURSOR ;
;           GA_NAAR_HEX_POSITIE, GA_NAAR_ASCII_POSITIE ;
;           SCHRIJF_FANTOOM, SCHRIJF_PROMPTREGEL, CURSOR_RECHTS ;
;           SCHRIJF_HEX, SCHRIJF_TEK, SCHRIJF_NAAR_GEHEUGEN ;
; Leest:    EDITOR_PROMPT ;
;-----;
EDIT_BYTE  PROC    NEAR
            PUSH    DX
            CALL    BEWAAR_ECHTE_CURSOR
            CALL    GA_NAAR_HEX_POSITIE      ;ga naar het hex-getal in het
            CALL    CURSOR_RECHTS           ; hex-venster
            CALL    SCHRIJF_HEX             ;schrijf het hex-getal
            CALL    GA_NAAR_ASCII_POSITIE   ;ga naar het teken in het ASCII-venster
            CALL    SCHRIJF_TEK             ;schrijf het nieuwe teken
            CALL    HERSTEL_ECHTE_CURSOR    ;zet cursor weer waar hij hoort
            CALL    SCHRIJF_FANTOOM         ;herschrijf de fantoomcursor
            CALL    SCHRIJF_NAAR_GEHEUGEN   ;bewaar deze nieuwe byte in SECTOR
            LEA     DX,EDITOR_PROMPT
            CALL    SCHRIJF_PROMPTREGEL
            POP     DX
            RET
EDIT_BYTE  ENDP
CODE_SEG   ENDS

END

```


B.2.6 FANTOOM.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP
```

```
CODE_SEG      SEGMENT PUBLIC
```

```
        PUBLIC  GA_NAAR_HEX_POSITIE
        EXTRN   GANAAR_XY:NEAR
DATA_SEG      SEGMENT PUBLIC
        EXTRN   REGELS_VOOR_SECTOR:BYTE
DATA_SEG      ENDS
```

```
-----;
; Deze procedure zet de echte cursor op de plaats van de fantoomcursor ;
; in het hex-venster. ;
; ;
; Gebruikt:      GANAAR_XY ;
; Leest:         REGELS_VOOR_SECTOR, FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
-----;
GA_NAAR_HEX_POSITIE  PROC    NEAR
        PUSH     AX
        PUSH     CX
        PUSH     DX
        MOV      DH,REGELS_VOOR_SECTOR ;zoek rij van fantoom (0,0)
        ADD      DH,2 ;plus rij van hex- en horizontale balk
        ADD      DH,FANTOOMCURSOR_Y ;DH = rij van fantoomcursor
        MOV      DL,8 ;inspringen linkerkant
        MOV      CL,3 ;elke kolom gebruikt 3 tekens, dus
        MOV      AL,FANTOOMCURSOR_X ;CURSOR_X moet met 3 vermenigvuldigd
        MUL      CL
        ADD      DL,AL ;en optellen bij inspringing, om kolom
        CALL     GANAAR_XY ;voor fantoomcursor te verkrijgen
        POP      DX
        POP      CX
        POP      AX
        RET
GA_NAAR_HEX_POSITIE  ENDP
```

```
        PUBLIC  GA_NAAR_ASCII_POSITIE
        EXTRN   GANAAR_XY:NEAR
DATA_SEG      SEGMENT PUBLIC
        EXTRN   REGELS_VOOR_SECTOR:BYTE
DATA_SEG      ENDS
```

```
-----;
; Deze procedure zet de echte cursor aan het begin van de fantoomcursor ;
; in het ASCII-venster. ;
; ;
; Gebruikt:      GANAAR_XY ;
; Leest:         REGELS_VOOR_SECTOR, FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
-----;
GA_NAAR_ASCII_POSITIE  PROC    NEAR
        PUSH     AX
        PUSH     DX
        MOV      DH,REGELS_VOOR_SECTOR ;zoek rij van fantoom (0,0)
        ADD      DH,2 ;plus rij van hex- en horizontale balk
        ADD      DH,FANTOOMCURSOR_Y ;DH = rij van fantoomcursor
```

FANTOOM.ASM *vervolg*

```

        MOV     DL,59                      ;inspringen linkerkant
        ADD     DL,FANTOOMCURSOR_X        ;tel CURSOR_X erbij op om positie
        CALL    GANAAR_XY                 ; X voor fantoomcursor te verkrijgen
        POP     DX
        POP     AX
        RET

GA_NAAR_ASCII_POSITIE    ENDP

        PUBLIC  BEWAAR_ECHTE_CURSOR

;-----;
; Deze procedure bewaart de plaats van de echte cursor in de twee ;
; variabelen ECHTE_CURSOR_X en ECHTE_CURSOR_Y.                      ;
;                                                                     ;
; Schrijft:      ECHTE_CURSOR_X, ECHTE_CURSOR_Y                    ;
;-----;
BEWAAR_ECHTE_CURSOR     PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                      ;lees cursorpositie
        XOR     BH,BH                     ; op pagina 0
        INT     10h                       ;en zet in DL,DH
        MOV     ECHTE_CURSOR_Y,DL        ;bewaar positie
        MOV     ECHTE_CURSOR_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
BEWAAR_ECHTE_CURSOR     ENDP

        PUBLIC  HERSTEL_ECHTE_CURSOR
        EXTRN   GANAAR_XY:NEAR

;-----;
; Deze procedure zet de echte cursor weer op zijn oude plaats, die is ;
; bewaard in ECHTE_CURSOR_X en ECHTE_CURSOR_Y.                      ;
;                                                                     ;
; Gebruikt:      GANAAR_XY                                           ;
; Leest:         ECHTE_CURSOR_X, ECHTE_CURSOR_Y                    ;
;-----;
HERSTEL_ECHTE_CURSOR     PROC    NEAR
        PUSH    DX
        MOV     DL,ECHTE_CURSOR_Y
        MOV     DH,ECHTE_CURSOR_X
        CALL    GANAAR_XY
        POP     DX
        RET
HERSTEL_ECHTE_CURSOR     ENDP

        PUBLIC  SCHRIJF_FANTOOM
        EXTRN   SCHRIJF_ATTRIBUUT_N_KEER:NEAR

```


FANTOOM.ASM *vervolg*

```

;-----;
; Deze procedure gebruikt CURSOR_X en CURSOR_Y, via GA_NAAR..., als de ;
; coördinaten voor de fantoomcursor. SCHRIJF_FANTOOM schrijft deze ;
; fantoomcursor. ;
; ;
; Gebruikt: SCHRIJF_ATTRIBUUT_N_KEER, BEWAAR_ECHTE_CURSOR ;
; HERSTEL_ECHTE_CURSOR, GA_NAAR_HEX_POSITIE ;
; GA_NAAR_ASCII_POSITIE ;
;-----;
SCHRIJF_FANTOOM PROC NEAR
    PUSH    CX
    PUSH    DX
    CALL    BEWAAR_ECHTE_CURSOR
    CALL    GA_NAAR_HEX_POSITIE      ;coörd. van cursor in hex-venster
    MOV     CX,4                    ;maak fantoomcursor vier tekens breed
    MOV     DL,70h
    CALL    SCHRIJF_ATTRIBUUT_N_KEER
    CALL    GA_NAAR_ASCII_POSITIE   ;coörd. van cursor in ASCII-venster
    MOV     CX,1                    ;cursor is hier een teken breed
    CALL    SCHRIJF_ATTRIBUUT_N_KEER
    CALL    HERSTEL_ECHTE_CURSOR
    POP     DX
    POP     CX
    RET
SCHRIJF_FANTOOM ENDP

    PUBLIC  WIS_FANTOOM
    EXTRN  SCHRIJF_ATTRIBUUT_N_KEER:NEAR

;-----;
; Deze procedure verwijdert de fantoomcursor - precies het ;
; tegenovergestelde van SCHRIJF_FANTOOM. ;
; ;
; Gebruikt: SCHRIJF_ATTRIBUUT_N_KEER, BEWAAR_ECHTE_CURSOR ;
; HERSTEL_ECHTE_CURSOR, GA_NAAR_HEX_POSITIE ;
; GA_NAAR_ASCII_POSITIE ;
;-----;
WIS_FANTOOM PROC NEAR
    PUSH    CX
    PUSH    DX
    CALL    BEWAAR_ECHTE_CURSOR
    CALL    GA_NAAR_HEX_POSITIE     ;coörd. van cursor in hex-venster
    MOV     CX,4                    ;verander weer naar wit op zwart
    MOV     DL,7
    CALL    SCHRIJF_ATTRIBUUT_N_KEER
    CALL    GA_NAAR_ASCII_POSITIE
    MOV     CX,1
    CALL    SCHRIJF_ATTRIBUUT_N_KEER
    CALL    HERSTEL_ECHTE_CURSOR
    POP     DX
    POP     CX
    RET
WIS_FANTOOM ENDP

```

FANTOOM.ASM *vervolg*

```

;-----;
; Deze vier procedures verplaatsen de fantoomcursors. ;
; ;
; Gebruikt: WIS_FANTOOM, SCHRIJF_FANTOOM ;
; SCHUIF_OMLAAG, SCHUIF_OMHOOG ;
; Leest: FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
; Schrijft: FANTOOMCURSOR_X, FANTOOMCURSOR_Y ;
;-----;

PUBLIC FANTOOM_OMHOOG
FANTOOM_OMHOOG PROC NEAR
CALL WIS_FANTOOM ;wis op huidige positie
DEC FANTOOMCURSOR_Y ;zet cursor een regel hoger
JNS STOND_NIET_BOVEN ;stond niet bovenaan, schrijf cursor
CALL SCHUIF_OMLAAG ;stond bovenaan, schuiven
STOND_NIET_BOVEN:
CALL SCHRIJF_FANTOOM ;schrijf fantoom op nieuwe positie
RET
FANTOOM_OMHOOG ENDP

PUBLIC FANTOOM_OMLAAG
FANTOOM_OMLAAG PROC NEAR
CALL WIS_FANTOOM ;wis op huidige positie
INC FANTOOMCURSOR_Y ;zet cursor een regel lager
CMP FANTOOMCURSOR_Y,16 ;stond hij onderaan?
JB STOND_NIET_ONDER ;nee, dus schrijf fantoom
CALL SCHUIF_OMHOOG ;stond onderaan, schuiven
STOND_NIET_ONDER:
CALL SCHRIJF_FANTOOM ;schrijf de fantoomcursor
RET
FANTOOM_OMLAAG ENDP

PUBLIC FANTOOM_LINKS
FANTOOM_LINKS PROC NEAR
CALL WIS_FANTOOM ;wis op huidige positie
DEC FANTOOMCURSOR_X ;zet cursor een kolom naar links
JNS STOND_NIET_LINKS ;stond niet links, schrijf cursor
MOV FANTOOMCURSOR_X,0 ;stond links, dus zet daar terug
STOND_NIET_LINKS:
CALL SCHRIJF_FANTOOM ;schrijf de fantoomcursor
RET
FANTOOM_LINKS ENDP

PUBLIC FANTOOM_RECHTS
FANTOOM_RECHTS PROC NEAR
CALL WIS_FANTOOM ;wis op huidige positie
INC FANTOOMCURSOR_X ;zet cursor een kolom naar rechts
CMP FANTOOMCURSOR_X,16 ;stond hij al aan rechterkant?
JB STOND_NIET_RECHTS ;stond rechts, dus zet daar terug
MOV FANTOOMCURSOR_X,15
STOND_NIET_RECHTS:
CALL SCHRIJF_FANTOOM ;schrijf de fantoomcursor
RET
FANTOOM_RECHTS ENDP

```


FANTOOM.ASM *vervolg*

```

        EXTRN    TOON_HALVE_SECTOR:NEAR, GANAAR_XY:NEAR
DATA_SEG    SEGMENT PUBLIC
        EXTRN    SECTOR_OFFSET:WORD
        EXTRN    REGELS_VOOR_SECTOR:BYTE
DATA_SEG    ENDS

;-----;
; Deze twee procedures wisselen tussen de twee halve sector-;
; afbeeldingen.                                           ;
;                                                         ;
; Gebruikt:        SCHRIJF_FANTOOM, TOON_HALVE_SECTOR, WIS_FANTOOM, ;
;                 GANAAR_XY, BEWAAR_ECHTE_CURSOR, HERSTEL_ECHTE_CURSOR ;
; Leest:          REGELS_VOOR_SECTOR                      ;
; Schrijft:       SECTOR_OFFSET, FANTOOMCURSOR_Y          ;
;-----;
SCHUIF_OMHOOG    PROC    NEAR
        PUSH    DX
        CALL    WIS_FANTOOM                ;verwijder fantoomcursor
        CALL    BEWAAR_ECHTE_CURSOR        ;bewaar positie echte cursor
        XOR     DL,DL                      ;stel cursor in voor halve sector-
                                           ; afbeelding

        MOV     DH,REGELS_VOOR_SECTOR
        ADD     DH,2
        CALL    GANAAR_XY
        MOV     DX,256                    ;beeld tweede helft sector af
        MOV     SECTOR_OFFSET,DX
        CALL    TOON_HALVE_SECTOR
        CALL    HERSTEL_ECHTE_CURSOR      ;herstel positie echte cursor
        MOV     FANTOOMCURSOR_Y,0         ;cursor bovenaan tweede halve sector
        CALL    SCHRIJF_FANTOOM           ;herstel fantoomcursor
        POP     DX
        RET
SCHUIF_OMHOOG    ENDP

SCHUIF_OMLAAG    PROC    NEAR
        PUSH    DX
        CALL    WIS_FANTOOM                ;verwijder fantoomcursor
        CALL    BEWAAR_ECHTE_CURSOR        ;bewaar positie echte cursor
        XOR     DL,DL                      ;stel cursor in voor halve sector-
                                           ; afbeelding

        MOV     DH,REGELS_VOOR_SECTOR
        ADD     DH,2
        CALL    GANAAR_XY
        XOR     DX,DX                    ;beeld tweede helft sector af
        MOV     SECTOR_OFFSET,DX
        CALL    TOON_HALVE_SECTOR
        CALL    HERSTEL_ECHTE_CURSOR      ;herstel positie echte cursor
        MOV     FANTOOMCURSOR_Y,15        ;cursor onderaan twee halve sector
        CALL    SCHRIJF_FANTOOM           ;herstel fantoomcursor
        POP     DX
        RET
SCHUIF_OMLAAG    ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC

```

FANTOOM.ASM *vervolg*

```
ECHTE_CURSOR_X      DB      0
ECHTE_CURSOR_Y      DB      0
      PUBLIC FANTOOMCURSOR_X, FANTOOMCURSOR_Y
FANTOOMCURSOR_X DB      0
FANTOOMCURSOR_Y DB      0
DATA_SEG      ENDS
```

END

B.2.7 TBD_IO.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP
```

```
BS      EQU     8           ;backspace-teken
CR      EQU     13          ;teken voor carriage return
ESC     EQU     27          ;teken voor Escape
```

```
CODE_SEG  SEGMENT PUBLIC
```

```
        PUBLIC  STRING_NAAR_HOOFDLET
```

```

;-----;
; Deze procedure zet de string, met het DOS-formaat voor strings, om in ;
; een reeks hoofdletters. ;
; ;
;      DS:DX  Adres van stringbuffer ;
;-----;
STRING_NAAR_HOOFDLET      PROC      NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX           ;wijs naar aantal tekens
        MOV     CL,[BX]      ;aantal tekens in 2de byte van buffer
        XOR     CH,CH        ;wis hoge byte van teken-telling
HOOFDLET_LUS:
        INC     BX           ;wijs naar volgende teken in buffer
        MOV     AL,[BX]
        CMP     AL,'a'       ;kijk of het een kleine letter is
        JB      NIET_KLEIN   ;nee
        CMP     AL,'z'
        JA      NIET_KLEIN
        ADD     AL,'A'-'a'    ;zet om in hoofdletter
        MOV     [BX],AL
NIET_KLEIN:
        LOOP    HOOFDLET_LUS
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_NAAR_HOOFDLET      ENDP
```

```

;-----;
; Deze procedure zet een teken van ASCII (hex) om in een nibble ;
; (4 bits). ;
; ;
;      AL      Om te zetten teken ;
; Geeft door:  AL      Nibble ;
;      DF      Ingesteld bij fout, anders nul ;
;-----;
CONVERTEER_HEX_CIJFER  PROC      NEAR
        CMP     AL,'0'       ;is 't een geldig cijfer?
        JB      ONGELDIG_CIJFER ;nee
        CMP     AL,'9'
        JA      PROBEER_HEX  ;nog niet zeker
        ;is misschien hex-cijfer
```

TBD_IO.ASM *vervolg*

```

        SUB     AL,'0'                ;is decimaal cijfer, zet om in nibble
        CLC
        RET                                ;zet carry uit, geen fout
PROBEER_HEX:
        CMP     AL,'A'                ;nog niet zeker
        JB      ONGELDIG_CIJFER        ;geen hex
        CMP     AL,'F'                ;nog niet zeker
        JA      ONGELDIG_CIJFER        ;geen hex
        SUB     AL,'A'-10              ;is hex, zet om in nibble
        CLC
        RET                                ;zet carry uit, geen fout
ONGELDIG_CIJFER:
        STC                                ;zet carry aan, fout
        RET
CONVERTEER_HEX_CIJFER    ENDP

        PUBLIC  HEX_NAAR_BYTE

;-----;
; Deze procedure zet de twee tekens in DS:DX van hex om in een byte. ;
;-----;
;      DS:DX    Adres van twee tekens voor hex-getal ;
; Geeft: ;
;      AL       Byte ;
;      CF       Bij fout aan, uit als geen fout ;
; Gebruikt:    CONVERTEER_HEX_CIJFER ;
;-----;
HEX_NAAR_BYTE    PROC    NEAR
        PUSH    BX
        PUSH    CX
        MOV     BX,DX                ;zet adres in BX voor indirect adres
        MOV     AL,[BX]              ;haal eerste cijfer
        CALL    CONVERTEER_HEX_CIJFER
        JC      ONGELDIG_HEX          ;onjuist hex-cijfer als carry aan
        MOV     CX,4                  ;nu vermenigvuldigen met 16
        SHL     AL,CL
        MOV     AH,AL
        INC     BX                    ;bewaar een kopie
        MOV     AL,[BX]              ;haal tweede cijfer
        CALL    CONVERTEER_HEX_CIJFER
        JC      ONGELDIG_HEX          ;onjuist hex-cijfer als carry aan
        OR      AL,AH                 ;combineer twee nibbles
        CLC                            ;zet carry uit als geen fout
KLAAR_HEX:
        POP     CX
        POP     BX
        RET
ONGELDIG_HEX:
        STC                                ;zet carry aan bij fout
        JMP     KLAAR_HEX
HEX_NAAR_BYTE    ENDP

        PUBLIC  LEES_STRING
        EXTRN   SCHRIJF_TEK:NEAR

```



```

;-----;
; Deze procedure verricht een functie die erg lijkt op de DOS-functie ;
; 0Ah. Maar deze functie geeft een speciaal teken door als een functie- ;
; toets of speciale toets wordt ingedrukt - geen carriage return. En ;
; ESC wist de invoer en begint opnieuw. ;
; ;
; DS:DX Adres voor toetsenbord-buffer. De eerste byte moet het ;
; maximum aantal te lezen tekens bevatten (plus een voor ;
; de return) En de tweede byte wordt door deze procedure ;
; gebruikt om het aantal feitelijk gelezen tekens door te ;
; geven. ;
; 0 Geen tekens gelezen ;
; -1 Een speciaal teken gelezen ;
; anders feitelijk gelezen aantal (uitgezonderd ;
; de Enter-toets) ;
; ;
; Gebruikt: BACKSPACE, SCHRIJF_TEK ;
;-----;
LEES_STRING PROC NEAR
    PUSH AX
    PUSH BX
    PUSH SI
    MOV SI,DX ;gebruik SI als indexregister en
START_OPNIEUW:
    MOV BX,2 ; BX als offset naar begin van buffer
    MOV AH,7 ;roep invoer zonder controle aan
    INT 21h ; voor CTRL-BREAK en geen echo
    OR AL,AL ;is teken uitgebreide ASCII?
    JZ UITGEBREID ;ja, lees uitgebreide teken
NIET_UTGEBREID:
    CMP AL,CR ;uitgebreide is fout tenzij buffer leeg
    JE EINDE_INVOER ;is dit een carriage return?
    CMP AL,BS ;ja, we zijn klaar met invoer
    JNE NIET_BS ;is het een backspace-teken?
    CALL BACKSPACE ;nee
    CMP BL,2 ;ja, verwijder teken
    JE START_OPNIEUW ;is buffer leeg?
    JMP SHORT LEES_VOLGENDE_TEK ;ja, nu kunnen we uitgebreide ASCII
NIET_BS: ; weer lezen
    CMP AL,ESC ;nee, vervolg lezen normale tekens
    JE LEDIG_BUFFER ;is het een ESC-leegmaakbuffer?
    CMP BL,[SI] ;ja, maak buffer leeg
    JA BUFFER_VOL ;controleer of buffer vol is
    MOV [SI+BX],AL ;buffer is vol
    INC BX ;bewaar teken anders in buffer
    ;wijst naar volgende vrije teken in
    ; buffer
    PUSH DX
    MOV DL,AL ;stuur teken ook naar scherm
    CALL SCHRIJF_TEK
    POP DX
LEES_VOLGENDE_TEK:
    MOV AH,7
    INT 21h
    OR AL,AL ;een uitgebreid ASCII-teken is ongeldig

```

TBD_IO.ASM *vervolg*

```

; als de buffer niet leeg is
JNE    NIET_UITGEBREID    ;teken is geldig
MOV     AH,7
INT     21h                ;verwerp uitgebreide teken

;-----;
; Geef een foutconditie aan door een pieptoon- ;
; teken naar het scherm te sturen: chr$(7).    ;
;-----;
FOUT_PIEP:
    PUSH    DX
    MOV     DL,7            ;geef pieptoon met chr$(7)
    MOV     AH,2
    INT     21h
    POP     DX
    JMP     SHORT LEES_VOLGENDE_TEK ;lees nu volgende teken

;-----;
; Maak de stringbuffer leeg en wis alle tekens ;
; die op het scherm staan afgebeeld.          ;
;-----;
LEDIG_BUFFER:
    PUSH    CX
    MOV     CL,[SI]        ;terugstellen over maximum aantal
    XOR     CH,CH
LEDIG_LUS:
    CALL    BACKSPACE      ; tekens in buffer. BACKSPACE voorkomt
    LOOP    LEDIG_LUS      ; dat de cursor te ver terug gaat
    POP     CX
    JMP     START_OPNIEUW   ;kan nu uitgebreide ASCII-tekens lezen
                                ; omdat de buffer leeg is

;-----;
; De buffer was vol, dus kan geen teken meer  ;
; lezen. Geef een pieptoon om gebruiker erop  ;
; te attenderen dat buffer vol is.            ;
;-----;
BUFFER_VOL:
    JMP     SHORT FOUT_PIEP    ;als buffer vol, alleen pieptoon geven

;-----;
; Lees de uitgebreide ASCII-CODE en zet die   ;
; als het enige teken in de buffer, geef dan ;
; -1 als het aantal gelezen tekens.           ;
;-----;
UITGEBREID:
                                ;lees een uitgebreide ASCII-code
    MOV     AH,7
    INT     21h
    MOV     [SI+2],AL        ;zet alleen dit teken in buffer
    MOV     BL,0FFh         ;aantal gelezen tekens = -1 voor
                                ; speciaal
    JMP     SHORT EINDE_STRING

```


TBD_IO.ASM *vervolg*

```

;-----;
; Bewaar getelde aantal gelezen tekens en geef ;
; door. ;
;-----;
EINDE_INVOER: ;klaar met invoer
              SUB     BL,2 ;gelezen aantal tekens
EINDE_STRING:
              MOV     [SI+1],BL ;geef aantal gelezen tekens door
              POP     SI
              POP     BX
              POP     AX
              RET
LEES_STRING   ENDP

PUBLIC LEES_BYTE
;-----;
; Deze procedure leest een enkel ASCII-teken of een hex-getal ;
; van twee cijfers. ;
; ;
; Zet byte in AL Code teken (tenzij AH = 0) ;
; AH 1 als ASCII-teken of hex-getal gelezen ;
; 0 als geen tekens gelezen ;
; -1 als een speciale toets gelezen ;
; ;
; Gebruikt: HEX_NAAR_BYTE, STRING_NAAR_HOOFDLET, LEES_STRING ;
; Leest: TOETSENBORD_INVOER, etc. ;
; Schrijft: TOETSENBORD_INVOER, etc. ;
;-----;
LEES_BYTE PROC NEAR
    PUSH DX
    MOV GRENS_AANTAL_TEK,3 ;sta twee tekens toe (plus Enter)
    LEA DX,TOETSENBORD_INVOER
    CALL LEES_STRING
    CMP GELEZEN_AANTAL_TEK,1 ;kijk hoe veel tekens
    JE ASCII_INVOER ;maar een, behandel als ASCII-teken
    JB GEEN_TEKENS ;alleen Enter-toets ingedrukt
    CMP BYTE PTR GELEZEN_AANTAL_TEK,0FFh ;speciale functietoets?
    JE SPECIALE_TOETS ;ja
    CALL STRING_NAAR_HOOFDLET ;nee, zet string om in hoofdletters
    LEA DX,TEKENS ;adres van om te zetten string
    CALL HEX_NAAR_BYTE ;zet string van hex om in byte
    JC GEEN_TEKENS ;fout, dus geef 'geen tekens gelezen'
    MOV AH,1 ;geef een teken gelezen door
KLAAR_MET_LEZEN:
    POP DX
    RET
GEEN_TEKENS:
    XOR AH,AH ;zet op 'geen tekens gelezen'
    JMP KLAAR_MET_LEZEN
ASCII_INVOER:
    MOV AL,TEKENS ;laad gelezen teken
    MOV AH,1 ;geef een teken gelezen door
    JMP KLAAR_MET_LEZEN
SPECIALE_TOETS:
    MOV AL,TEKENS[0] ;geef scancode door
    MOV AH,0FFh ;attendeer met -1 op speciale toets

```

```

        JMP      KLAAR_MET_LEZEN
LEES_BYTE ENDP

```

```

        PUBLIC   LEES_DECIMAAL
;-----;
; Deze procedure neemt de uitvoerbuffer van LEES_STRING en zet de
; reeks decimale cijfers om in een woord.
;
;      AX      Van decimaal omgezet woord
;      CF      Aan als fout, uit als geen fout
;
; Gebruikt:    LEES_STRING
; Leest:       TOETSENBORD_INVOER, etc.
; Schrijft:    TOETSENBORD_INVOER, etc.
;-----;
LEES_DECIMAAL PROC NEAR
        PUSH     BX
        PUSH     CX
        PUSH     DX
        MOV      GRENS_AANTAL_TEK,6      ;maximaal 5 cijfers (65535)
        LEA      DX,TOETSENBORD_INVOER
        CALL     LEES_STRING
        MOV      CL,GELEZEN_AANTAL_TEK  ;vraag aantal gelezen tekens op
        XOR      CH,CH                  ;maak hoge byte van telling 0
        CMP      CL,0                   ;geef fout als geen tekens gelezen
        JLE      ONGELDIGE_DECIMAAL     ;geen tekens gelezen, geef fout
        XOR      AX,AX                  ;begin met getal op 0
        XOR      BX,BX                  ;begin bij begin string
CONVERTEER_CIJFER:
        MOV      DX,10                  ;vermenigvuldig getal met 10
        MUL      DX                      ;vermenigvuldig AX met 10
        JC       ONGELDIGE_DECIMAAL     ;CF aan als overloop MUL uit een woord
        MOV      DL,TEKENS[BX]          ;vraag volgende cijfer op
        SUB      DL,'0'                 ;en zet om in een nibble (4 bits)
        JS       ONGELDIGE_DECIMAAL     ;onjuist cijfer als < 0
        CMP      DL,9                   ;is dit een onjuist cijfer?
        JA       ONGELDIGE_DECIMAAL     ;ja
        ADD      AX,DX                  ;nee, tel dan op bij getal
        INC      BX                     ;wijs naar volgende teken
        LOOP     CONVERTEER_CIJFER      ;vraag volgende cijfer op
KLAAR_DECIMAAL:
        POP      DX
        POP      CX
        POP      BX
        RET
ONGELDIGE_DECIMAAL:
        STC                                ;zet carry op fout aangeven
        JMP      KLAAR_DECIMAAL
LEES_DECIMAAL ENDP

        PUBLIC   BACKSPACE
        EXTRN    SCHRIJF_TEK:NEAR

```


TBD_IO.ASM *vervolg*

```

;-----;
; Deze procedure wist tekens, met een tegelijk, uit de buffer en van het;
; het scherm wanneer die buffer niet leeg is. BACKSPACE keert gewoon ;
; terug als de buffer leeg is. ;
; ;
; DS:SI+BX Laatst getikte teken dat nog in buffer staat ;
; ;
; Gebruikt: SCHRIJF_TEK ;
;-----;
BACKSPACE PROC NEAR ;wis een teken
    PUSH AX
    PUSH DX
    CMP BX,2 ;is buffer leeg?
    JE EINDE_BS ;ja, lees volgende teken
    DEC BX ;haal een teken uit buffer
    MOV AH,2 ;haal teken van scherm
    MOV DL,BS
    INT 21h
    MOV DL,20h ;schrijf daar spatie
    CALL SCHRIJF_TEK
    MOV DL,BS ;weer terug
    INT 21h
EINDE_BS:
    POP DX
    POP AX
    RET
BACKSPACE ENDP

CODE_SEG ENDS

DATA_SEG SEGMENT PUBLIC
TOETSENBOARD_INVOER LABEL BYTE
GRENS_AANTAL_TEK DB 0 ;lengte van invoerbuffer
GELEZEN_AANTAL_TEK DB 0 ;aantal gelezen tekens
TEKENS DB 80 DUP (0) ;een buffer voor toetsenbord-invoer
DATA_SEG ENDS

END

```

B.2.8 TOON_SEC.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG      ;voeg twee segmenten samen
        ASSUME  CS:CGROEP, DS:CGROEP

;-----;
; Grafische tekens voor randen van sector.
;-----;
VERTICALE_BALK      EQU    0BAh
HORIZONTALE_BALK    EQU    0CDh
BOVEN_LINKS         EQU    0C9h
BOVEN_RECHTS        EQU    0BBh
ONDER_LINKS          EQU    0C8h
ONDER_RECHTS         EQU    0BCh
TOP_T_BALK           EQU    0CBh
BENEDEN_T_BALK       EQU    0CAh
TOP_STREEPJE         EQU    0D1h
BENEDEN_STREEPJE     EQU    0CFh

CODE_SEG            SEGMENT PUBLIC

        PUBLIC   START_SEC_AFB
        EXTRN    SCHRIJF_PATROON:NEAR, STUUR_CRLF:NEAR
        EXTRN    GANAAR_XY:NEAR

;-----;
; Deze procedure start de halve sector-afbeelding.
;-----;
;
; Gebruikt:      SCHRIJF_PATROON, STUUR_CRLF, TOON_HALVE_SECTOR
;
;               SCHRIJF_HEX_GETALLEN_BOVENLANGS, GANAAR_XY
;
; Leest:         BOVENSTE_REGEL_PATROON, ONDERSTE_REGEL_PATROON
;-----;
START_SEC_AFB       PROC     NEAR
        PUSH     DX
        XOR      DL,DL                ;zet cursor op plaats aan begin
        MOV      DH,2                ; van 3de regel
        CALL     GANAAR_XY
        CALL     SCHRIJF_HEX_GETALLEN_BOVENLANGS
        LEA      DX,BOVENSTE_REGEL_PATROON
        CALL     SCHRIJF_PATROON
        CALL     STUUR_CRLF
        XOR      DX,DX                ;begin bij begin van de sector
        CALL     TOON_HALVE_SECTOR
        LEA      DX,ONDERSTE_REGEL_PATROON
        CALL     SCHRIJF_PATROON
        POP      DX
        RET
START_SEC_AFB       ENDP

        EXTRN    SCHRIJF_TEK_N_KEER:NEAR, SCHRIJF_HEX:NEAR, SCHRIJF_TEK:NEAR
        EXTRN    SCHRIJF_HEX_CIJFER:NEAR, STUUR_CRLF:NEAR

;-----;
; Deze procedure schrijft de getallen 0 t/m F bovenaan de halve
; sector-afbeelding
;-----;
;
; Gebruikt:      SCHRIJF_TEK_N_KEER, SCHRIJF_HEX, SCHRIJF_TEK
;
;               SCHRIJF_HEX_CIJFER, STUUR_CRLF
;-----;

```


TOON_SEC.ASM *vervolg*

```

SCHRIJF_HEX_GETALLEN_BOVENLANGS PROC    NEAR
    PUSH    CX
    PUSH    DX
    MOV     DL,' '                      ;schrijf 9 spaties voor linkerkant
    MOV     CX,9
    CALL    SCHRIJF_TEK_N_KEER
    XOR     DH,DH                      ;begin met 0
HEX_GETAL_LUS:
    MOV     DL,DH
    CALL    SCHRIJF_HEX
    MOV     DL,' '
    CALL    SCHRIJF_TEK
    INC     DH
    CMP     DH,10h                    ;klaar?
    JB      HEX_GETAL_LUS

    MOV     DL,' '                      ;schrijf hex-getallen boven ASCII-venster
    MOV     CX,2
    CALL    SCHRIJF_TEK_N_KEER
    XOR     DL,DL
HEX_CIJFER_LUS:
    CALL    SCHRIJF_HEX_CIJFER
    INC     DL
    CMP     DL,10h
    JB      HEX_CIJFER_LUS
    CALL    STUUR_CRLF
    POP     DX
    POP     CX
    RET
SCHRIJF_HEX_GETALLEN_BOVENLANGS ENDP

    PUBLIC  TOON_HALVE_SECTOR
    EXTRN  STUUR_CRLF:NEAR

;-----;
; Deze procedure beeldt een halve sector (256 bytes) af. ;
; ;
; DS:DX Offset in sector, in bytes -- moet meervoud van 16 zijn ;
; ;
; Gebruikt: TOON_REGEL, STUUR_CRLF ;
;-----;
TOON_HALVE_SECTOR PROC    NEAR
    PUSH    CX
    PUSH    DX
    MOV     CX,16                      ;beeld 16 regels af
HALVE_SECTOR:
    CALL    TOON_REGEL
    CALL    STUUR_CRLF
    ADD     DX,16
    LOOP    HALVE_SECTOR
    POP     DX
    POP     CX
    RET
TOON_HALVE_SECTOR ENDP

    PUBLIC  TOON_REGEL

```

TOON_SEC.ASM *vervolg*

```

        EXTRN    SCHRIJF_HEX:NEAR
        EXTRN    SCHRIJF_TEK:NEAR
        EXTRN    SCHRIJF_TEK_N_KEER:NEAR
;-----;
; Deze procedure drukt een regel met gegevens, of 16 bytes, af; eerst ;
; in hex, daarna in ASCII. ;
; ;
; DS:DX   Offset in sector, in bytes. ;
; ;
; Gebruikt: SCHRIJF_TEK, SCHRIJF_HEX, SCHRIJF_TEK_N_KEER ;
; Leest:    SECTOR ;
;-----;
TOON_REGEL    PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                ;offset is nuttiger in BX
        MOV     DL,' '              ;schrijf offset in hex
        MOV     CX,3                ;schrijf 3 spaties voor regel
        CALL    SCHRIJF_TEK_N_KEER
        CMP     BX,100h              ;is het eerste cijfer een 1?
        JB      SCHRIJF_EEN         ;nee, schrijf spatie die al in DL staat
        MOV     DL,'1'              ;ja, zet dan '1' in DL voor uitvoer
SCHRIJF_EEN:
        CALL    SCHRIJF_TEK
        MOV     DL,BL                ;kopieer lage byte naar DL voor
                                      ;hex-uitvoer

        CALL    SCHRIJF_HEX
        MOV     DL,' '
        CALL    SCHRIJF_TEK
        MOV     DL,VERTECALE_BALK   ;teken linkerkant kader
        CALL    SCHRIJF_TEK
        MOV     DL,' '
        CALL    SCHRIJF_TEK

                                      ;schrijf nu 16 bytes
        MOV     CX,16                ;dump 16 bytes
        PUSH    BX                  ;bewaar de offset voor ASCII_LUS
HEX_LUS:
        MOV     DL,SECTOR[BX]        ;haal 1 byte op
        CALL    SCHRIJF_HEX          ;dump deze byte in hex
        MOV     DL,' '              ;zet een spatie tussen getallen
        CALL    SCHRIJF_TEK
        INC     BX
        LOOP    HEX_LUS
        MOV     DL,VERTECALE_BALK   ;schrijf verticale balk
        CALL    SCHRIJF_TEK
        MOV     DL,' '
        CALL    SCHRIJF_TEK
        MOV     CX,16
        POP     BX                  ;haal offset in SECTOR terug
ASCII_LUS:
        MOV     DL,SECTOR[BX]
        CALL    SCHRIJF_TEK
        INC     BX
        LOOP    ASCII_LUS

```


TOON_SEC.ASM *vervolg*

```

        MOV     DL,' '                ;teken rechterkant kader
        CALL    SCHRIJF_TEK
        MOV     DL,VERTICALE_BALK
        CALL    SCHRIJF_TEK
        POP     DX
        POP     CX
        POP     BX
        RET
TOON_REGEL      ENDP

CODE_SEG      ENDS

```

```

DATA_SEG      SEGMENT PUBLIC
               EXTRN  SECTOR:BYTE
BOVENSTE_REGEL_PATROON LABEL BYTE
        DB      ' ',7
        DB      BOVEN_LINKS,1
        DB      HORIZONTALE_BALK,12
        DB      TOP_STREEPJE,1
        DB      HORIZONTALE_BALK,11
        DB      TOP_STREEPJE,1
        DB      HORIZONTALE_BALK,11
        DB      TOP_STREEPJE,1
        DB      HORIZONTALE_BALK,12
        DB      TOP_T_BALK,1
        DB      HORIZONTALE_BALK,18
        DB      BOVEN_RECHTS,1
        DB      0
ONDERSTE_REGEL_PATROON LABEL BYTE
        DB      ' ',7
        DB      ONDER_LINKS,1
        DB      HORIZONTALE_BALK,12
        DB      BENEDEN_STREEPJE,1
        DB      HORIZONTALE_BALK,11
        DB      BENEDEN_STREEPJE,1
        DB      HORIZONTALE_BALK,11
        DB      BENEDEN_STREEPJE,1
        DB      HORIZONTALE_BALK,12
        DB      BENEDEN_T_BALK,1
        DB      HORIZONTALE_BALK,18
        DB      ONDER_RECHTS,1
        DB      0
DATA_SEG      ENDS

        END

```

B.2.9 VERDEEL.ASM

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROEP, DS:CGROEP
```

```
CODE_SEG      SEGMENT PUBLIC
```

```
        PUBLIC VERDELER
        EXTRN  LEES_BYTE:NEAR, EDIT_BYTE:NEAR;
        EXTRN  SCHRIJF_PROMPTREGEL:NEAR
DATA_SEG      SEGMENT PUBLIC
        EXTRN  EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
```

```
;-----;
; Dit is de centrale coördinator. Bij normaal wijzigen (editen) en ;
; bekijken, leest deze procedure tekens van het toetsenbord en, als het ;
; teken een opdrachttoets (b.v. een cursortoets) is, roept VERDELER ;
; procedures aan die het eigenlijke werk doen. Dit gebeurt bij speciale ;
; toetsen die vermeld staan in de tabel VERDEEL_TABEL, waarin de ;
; adressen van de procedures vlak na de toetsnamen zijn opgeslagen. ;
; Als het teken geen speciale toets is, moet het rechtstreeks in de ;
; sectorbuffer worden gezet -- dit is de edit-modus. ;
; ;
; Gebruikt:      LEES_BYTE, EDIT_BYTE ;
; Leest:         EDITOR_PROMPT ;
;-----;

VERDELER      PROC      NEAR
        PUSH      AX
        PUSH      BX
        PUSH      DX
VERDEEL_LUS:
        CALL      LEES_BYTE      ;lees teken naar AX
        OR        AH,AH          ;AX = 0 als geen teken gelezen, -1
                                ; voor uitgebreide code
        JZ        GEEN_TEKENS_GELEZEN ;geen teken gelezen, probeer opnieuw
        JS        SPECIALE_TOETS    ;lees uitgebreide code
        MOV       DL,AL
        CALL      EDIT_BYTE      ;was normaal teken, wijzig byte
        JMP       VERDEEL_LUS    ;lees nog een teken
SPECIALE_TOETS:
        CMP       AL,68          ;F10--stop?
        JE        EINDE_VERDEEL  ;ja, einde
                                ;gebruik BX om tabel door te kijken
        LEA       BX,VERDEEL_TABEL
SPECIALE_LUS:
        CMP       BYTE PTR [BX],0 ;einde van tabel?
        JE        NIET_IN_TABEL   ;ja, toets stond niet in tabel
        CMP       AL,[BX]         ;is het deze tabel-ingang?
        JE        VERDEEL        ;ja, dan werk verdelen
        ADD       BX,3            ;nee, probeer volgende ingang
        JMP       SPECIALE_LUS    ;controleer volgende tabel-ingang

VERDEEL:
        INC       BX              ;wijs naar adres van procedure
        CALL      WORD PTR [BX]   ;roep procedure aan
        JMP       VERDEEL_LUS     ;wacht op een andere toets
```


VERDEEL.ASM *vervolg*

```
NIET_IN_TABEL:                                ;doe niets, lees gewoon volgende teken
      JMP      VERDEEL_LUS
```

```
GEEN_TEKENS_GELEZEN:
      LEA      DX,EDITOR_PROMPT
      CALL     SCHRIJF_PROMPTREGEL      ;wis getikte tekens die ongeldig zijn
      JMP      VERDEEL_LUS             ;probeer opnieuw
```

```
EINDE_VERDEEL:
      POP      DX
      POP      BX
      POP      AX
      RET
```

```
VERDELER      ENDP
```

```
CODE_SEG      ENDS
```

```
DATA_SEG      SEGMENT PUBLIC
```

```
CODE_SEG      SEGMENT PUBLIC
      EXTRN    VOLGENDE_SECTOR:NEAR      ;in DISK_IO.ASM
      EXTRN    VORIGE_SECTOR:NEAR        ;in DISK_IO.ASM
      EXTRN    FANTOOM_OMHOOG:NEAR, FANTOOM_OMLAAG:NEAR;in FANTOOM.ASM
      EXTRN    FANTOOM_LINKS:NEAR, FANTOOM_RECHTS:NEAR
      EXTRN    SCHRIJF_SECTOR:NEAR      ;in DISK_IO.ASM
```

```
CODE_SEG      ENDS
```

```
;-----;
; Deze tabel bevat de toegestane uitgebreide ASCII-toetsen en de ;
; adressen van de procedures die moeten worden aangeroepen wanneer een ;
; toets wordt ingedrukt. ;
; De tabel heeft de volgende indeling: ;
;          DB 72 ;uitgebreide code voor cursor omhoog ;
;          DW OFFSET CGROEP:FANTOOM_OMHOOG ;
;-----;
```

```
VERDEEL_TABEL LABEL BYTE
      DB 59 ;F1
      DW OFFSET CGROEP:VORIGE_SECTOR
      DB 60 ;F2
      DW OFFSET CGROEP:VOLGENDE_SECTOR
      DB 72 ;cursor omhoog
      DW OFFSET CGROEP:FANTOOM_OMHOOG
      DB 80 ;cursor omlaag
      DW OFFSET CGROEP:FANTOOM_OMLAAG
      DB 75 ;cursor naar links
      DW OFFSET CGROEP:FANTOOM_LINKS
      DB 77 ;cursor naar rechts
      DW OFFSET CGROEP:FANTOOM_RECHTS
      DB 88 ;Shift-F5
      DW OFFSET CGROEP:SCHRIJF_SECTOR
      DB 0 ;einde van de tabel
```

```
DATA_SEG      ENDS
```

```
END
```

B.2.10 VIDEO_IO.ASM

```

CGROEP  GROUP  CODE_SEG, DATA_SEG      ;voeg twee segmenten
                                           ;samen
        ASSUME  CS:CGROEP

CODE_SEG      SEGMENT PUBLIC

        PUBLIC  SCHRIJF_PATROON

;-----;
; Deze procedure schrijft een regel naar het scherm, op basis van
; gegevens in de vorm
;
;      DB      {teken, aantal keren dat teken moet worden geschreven}, 0
; waarbij {x} betekent dat x een willekeurig aantal keren kan worden herhaald.
;      DS:DX    Adres van bovengenoemd datasegment
;
; Gebruikt:     SCHRIJF_TEK_N_KEER
;-----;
SCHRIJF_PATROON PROC    NEAR
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSHF                    ;bewaar richtingsvlag
    CLD                    ;stel richtingsvlag in t.b.v ophoging
    MOV     SI,DX           ;zet offset in SI-register voor LODSB
PATROON_LUS:
    LODSB                    ;haal teken-gegevens naar AL
    OR      AL,AL           ;is 't het einde van gegevens (0h)?
    JZ      EINDE_PATROON  ;ja, keer terug
    MOV     DL,AL           ;nee, klaarmaken voor N keer schrijven
    ; van teken
    LODSB                    ;zet herhaling-teller in AL
    MOV     CL,AL           ;en in CX t.b.v SCHRIJF_TEK_N_KEER
    XOR     CH,CH           ;hoge byte van CX nul maken
    CALL    SCHRIJF_TEK_N_KEER
    JMP     PATROON_LUS
EINDE_PATROON:
    POPF                    ;herstel richtingsvlag
    POP     SI
    POP     DX
    POP     CX
    POP     AX
    RET
SCHRIJF_PATROON ENDP
        PUBLIC  SCHRIJF_HEX

;-----;
; Deze procedure zet de byte in het DL-register om in hex en schrijft
; de twee hex-cijfers naar de huidige cursorpositie.
;
;      DL      Naar hex om te zetten byte.
;
; Gebruikt:     SCHRIJF_HEX_CIJFER
;-----;

```


VIDEO_IO.ASM *vervolg*

```

SCHRIJF_HEX      PROC    NEAR
                  PUSH    CX                ;ingangspunt
                                          ;bewaar in deze procedure gebruikte
                                          ; registers
                  PUSH    DX
                  MOV     DH,DL              ;maak kopie van byte
                  MOV     CX,4              ;zet hoogste nibble in DL
                  SHR     DL,CL
                  CALL    SCHRIJF_HEX_CIJFER ;druk eerste hex-cijfer af
                  MOV     DL,DH            ;zet laagste nibble in DL
                  AND     DL,0Fh           ;verwijder hoogste nibble
                  CALL    SCHRIJF_HEX_CIJFER ;druk twee hex-cijfer af
                  POP     DX
                  POP     CX
                  RET
SCHRIJF_HEX      ENDP

```

PUBLIC SCHRIJF_HEX_CIJFER

```

;-----;
; Deze procedure zet de laagste 4 bits van DL om in een hex-cijfer en ;
; schrijft het naar het scherm. ;
; ;
; DL Laagste 4 bits bevatten in hex af te drukken getal. ;
; ;
; Gebruikt: SCHRIJF_TEK ;
;-----;

```

```

SCHRIJF_HEX_CIJFER      PROC    NEAR
                  PUSH    DX                ;bewaar gebruikte registers
                  CMP     DL,10            ;is deze nibble <10?
                  JAE     HEX_LETTER       ;nee, zet om in letter
                  ADD     DL,"0"           ;ja, zet om in cijfer
                  JMP     Short SCHRIJF_CIJFER ;schrijf nu dit teken
HEX_LETTER:
                  ADD     DL,"A"-10        ;zet om in hex-letter
SCHRIJF_CIJFER:
                  CALL    SCHRIJF_TEK     ;druk letter af op scherm
                  POP     DX              ;zet oude waarde van AX terug
                  RET
SCHRIJF_HEX_CIJFER      ENDP

```

PUBLIC SCHRIJF_TEK
EXTRN CURSOR_RECHTS:NEAR

```

;-----;
; Deze procedure schrijft een teken naar het scherm m.b.v de ROM BIOS- ;
; routines, zodat tekens als backspace enz worden behandeld als elk ;
; ander teken en worden afgebeeld. ;
; Deze procedure moet eerst wat werk verrichten om de cursorpositie ;
; bij te werken. ;
; ;
; DL Byte die op het scherm moet worden afgedrukt ;
; ;
; Gebruikt: CURSOR_RECHTS ;
;-----;
SCHRIJF_TEK      PROC    NEAR
                  PUSH    AX

```

VIDEO_IO.ASM *vervolg*

```

        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,9                ;uitvoer teken/attribuut aanroepen
        MOV     BH,0                ;instellen opschermpagina 0
        MOV     CX,1                ;schrijf maar een teken
        MOV     AL,DL               ;te schrijven teken
        MOV     BL,7                ;normaal attribuut
        INT     10h                 ;schrijf teken en attribuut
        CALL    CURSOR_RECHTS       ;nu naar volgende cursorpositie
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SCHRIJF_TEK      ENDP

        PUBLIC  SCHRIJF_TEK_N_KEER
;-----;
; Deze procedure schrijft meer dan een kopie van een teken. ;
; ; ;
;     DL      Code teken ;
;     CX      Aantal keren dan teken moet worden geschreven ;
; ; ;
; Gebruikt:    SCHRIJF_TEK ;
;-----;
SCHRIJF_TEK_N_KEER  PROC  NEAR
        PUSH    CX
N_KEER:
        CALL    SCHRIJF_TEK
        LOOP    N_KEER
        POP     CX
        RET
SCHRIJF_TEK_N_KEER  ENDP

        PUBLIC  SCHRIJF_DECIMAAL
;-----;
; Deze procedure schrijft een 16-bits getal zonder teken in decimale ;
; notatie. ;
; ; ;
;     DX      N : 16-bits getal zonder teken. ;
; ; ;
; Gebruikt:    SCHRIJF_HEX_CIJFER ;
;-----;
SCHRIJF_DECIMAAL  PROC  NEAR
        PUSH    AX                ;bewaar hier gebruikte registers
        PUSH    CX
        PUSH    DX
        PUSH    SI
        MOV     AX,DX              ;zet het getal in AX
        MOV     SI,10              ;deelt door 10 met gebruik van SI
        XOR     CX,CX              ;aantal cijfers op stapel
NIET_NUL:
        XOR     DX,DX              ;maakt hoogste woord van N gelijk aan 0
        DIV     SI                 ;bereken N/10 en (N mod 10)

```


VIDEO_IO.ASM *vervolg*

```

        PUSH    DX                      ;zet een cijfer op de stapel
        INC     CX                      ;nog een cijfer erbij
        OR      AX,AX                  ;is N al 0?
        JNE     NIET_NUL               ;nee, doorgaan
SCHRIJF_CIJFER_LUS:
        POP     DX                      ;haal cijfers in omgekeerde volgorde op
        CALL    SCHRIJF_HEX_CIJFER
        LOOP    SCHRIJF_CIJFER_LUS
EINDE_DECIMAAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
SCHRIJF_DECIMAAL      ENDP

        PUBLIC  SCHRIJF_KOP
DATA_SEG      SEGMENT PUBLIC
        EXTRN  KOPREGEL_NR:BYTE
        EXTRN  KOP_DEEL_1:BYTE
        EXTRN  KOP_DEEL_2:BYTE
        EXTRN  DISKDRIVE_NR:BYTE
        EXTRN  HUIDIGE_SECTORNR:WORD
DATA_SEG      ENDS
        EXTRN  GANAAR_XY:NEAR, WIS_TOT_EINDE_REGEL:NEAR
;-----;
; Deze procedure schrijft de kop met diskdrive- en sector-nummer. ;
; ;
; Gebruikt:      GANAAR_XY, SCHRIJF_STRING, SCHRIJF_TEK, SCHRIJF_DECIMAAL;
;               WIS_TOT_EINDE_REGEL ;
; Leest:        KOPREGEL_NR, KOP_DEEL_1, KOP_DEEL_2 ;
;               DISKDRIVE_NR, HUIDIGE_SECTORNR ;
;-----;
SCHRIJF_KOP      PROC    NEAR
        PUSH    DX
        XOR     DL,DL                  ;zet cursor op regelnummer kop
        MOV     DH,KOPREGEL_NR
        CALL    GANAAR_XY
        LEA     DX,KOP_DEEL_1
        CALL    SCHRIJF_STRING
        MOV     DL,DISKDRIVE_NR
        ADD     DL,'A'                 ;druk drives A, B, ... af
        CALL    SCHRIJF_TEK
        LEA     DX,KOP_DEEL_2
        CALL    SCHRIJF_STRING
        MOV     DX,HUIDIGE_SECTORNR
        CALL    SCHRIJF_DECIMAAL
        CALL    WIS_TOT_EINDE_REGEL    ;wis rest van sectornummer
        POP     DX
        RET
SCHRIJF_KOP      ENDP

        PUBLIC  SCHRIJF_STRING

```

VIDEO_IO.ASM *vervolg*

```

;-----;
; Deze procedure schrijft een tekenstring naar het scherm. De string ;
; moet eindigen op          DB      0 ;
; ; ;
;      DS:DX  Adres van de string ;
; ; ;
; Gebruikt:      SCHRIJF_TEK ;
;-----;
SCHRIJF_STRING  PROC      NEAR
    PUSH      AX
    PUSH      DX
    PUSH      SI
    PUSHF
    CLD ;bewaar richtingvlag
           ;stel richting in op ophogen
           ; (naar voren)
    MOV      SI,DX ;zet adres in SI voor LODSB
STRING_LUS:
    LODSB ;zet een teken in het AL-register
    OR      AL,AL ;hebben we de 0 al gevonden?
    JZ      EINDE_STRING ;ja, klaar met de string
    MOV      DL,AL ;nee, schrijf teken
    CALL     SCHRIJF_TEK
    JMP      STRING_LUS
EINDE_STRING:
    POPF ;herstel richtingvlag
    POP      SI
    POP      DX
    POP      AX
    RET
SCHRIJF_STRING  ENDP

    PUBLIC    SCHRIJF_PROMPTREGEL
    EXTRN     WIS_TOT_EINDE_REGEL:NEAR
    EXTRN     GANAAR_XY:NEAR
DATA_SEG      SEGMENT PUBLIC
    EXTRN     PROMPTREGEL_NR:BYTE
DATA_SEG      ENDS

;-----;
; Deze procedure schrijft de promptregel naar het scherm en wist tot ;
; het einde van de regel. ;
; ; ;
;      DS:DX  Adres van de promptregel-melding ;
; ; ;
; Gebruikt:      SCHRIJF_STRING, WIS_TOT_EINDE_REGEL, GANAAR_XY ;
; Leest:         PROMPTREGEL_NR ;
;-----;
SCHRIJF_PROMPTREGEL  PROC      NEAR
    PUSH      DX
    XOR      DL,DL ;schrijf de promptregel en
    MOV      DH,PROMPTREGEL_NR ; zet daar de cursor heen
    CALL     GANAAR_XY
    POP      DX
    CALL     SCHRIJF_STRING
    CALL     WIS_TOT_EINDE_REGEL
    RET

```


VIDEO_IO.ASM *vervolg*

SCHRIJF_PROMPTREGEL ENDP

 PUBLIC SCHRIJF_ATTRIBUUT_N_KEER
 EXTRN CURSOR_RECHTS:NEAR

```

;-----;
; Deze procedure stelt het attribuut in op N tekens, vanaf de huidige ;
; cursorpositie. ;
; ;
;     CX     Aantal tekens waarvoor attribuut moet worden ingesteld ;
;     DL     Nieuw attribuut voor tekens ;
; ;
; Gebruikt:             CURSOR_RECHTS ;
;-----;
SCHRIJF_ATTRIBUUT_N_KEER                   PROC    NEAR
      PUSH    AX
      PUSH    BX
      PUSH    CX
      PUSH    DX
      MOV     BL,DL                         ;stel attribuut in op nieuw attribuut
      XOR     BH,BH                         ;stel schermpagina in op 0
      MOV     DX,CX                         ;CX wordt gebruikt door BIOS-routines
      MOV     CX,1                         ;stel attribuut voor een teken in
ATTR_LUS:
      MOV     AH,8                         ;lees teken onder cursor
      INT     10h
      MOV     AH,9                         ;schrijf attribuut/teken
      INT     10h
      CALL    CURSOR_RECHTS
      DEC     DX                            ;attribuut voor N tekens instellen?
      JNZ     ATTR_LUS                     ;nee, ga door
      POP     DX
      POP     CX
      POP     BX
      POP     AX
      RET
SCHRIJF_ATTRIBUUT_N_KEER                   ENDP

CODE_SEG           ENDS

      END

```

Appendix C

Laadvolgorde segmenten

- C.1 Laadvolgorde segmenten 350**
- C.2 Phase errors 352**
- C.3 EXE2BIN File cannot be converted 354**

De IBM Macro Assembler (versie 1.0 en 2.0) laadt segmenten in een andere volgorde in dan de recentere versies van de Microsoft Macro Assembler. In deze appendix bekijken we de kwestie van de volgorde waarin segmenten worden geladen, en zien we hoe kennis van deze volgorde van pas kan komen wanneer EXE2BIN u de melding File cannot be converted geeft.

C.1 Laadvolgorde segmenten

Bij alle voorbeelden na hoofdstuk 13 worden twee segmenten gebruikt, CODE_SEG en DATA_SEG. De IBM-versie van de assembler zegt LINK dat hij deze segmenten in alfabetische volgorde in het geheugen moet laden. Dus als we schrijven:

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

DATA_SEG      SEGMENT PUBLIC
DATA_SEG      ENDS

CODE_SEG      SEGMENT PUBLIC
CODE_SEG      ENDS

        END
```

vertellen de IBM-versies van de Macro Assembler LINK dat DATA_SEG ná CODE_SEG in het geheugen moet worden geladen. Laten we dit code-fragment eens in een echt programma omzetten om het laadoverzicht te kunnen bekijken. Dit is onze nieuwe versie. Veel doet ze niet, maar het is voldoende om te laten zien hoe LINK de segmenten in het geheugen laadt.

```
CGROEP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROEP, DS:CGROEP

DATA_SEG      SEGMENT PUBLIC
        DB      0
DATA_SEG      ENDS

CODE_SEG      SEGMENT PUBLIC
        ORG     100h
HOOFD:  INT     20h
CODE_SEG      ENDS

        END     HOOFD
```

Tik dit bestand in, noem het SEGTEST.ASM, assembleer het, en link het zo dat u een laadoverzicht krijgt:

```
C>LINK SEGTEST,SEGTEST,SEGTEST/MAP;
```

Als u een IBM-versie van de assembler hebt, ziet u dit laadoverzicht:

Warning: no stack segment

Start	Stop	Length	Name	Class
00000H	00101H	00102H	CODE_SEG	
00110H	00110H	00001H	DATA_SEG	

Origin	Group
0000:0	CGROEP

Address	Publics by Name
---------	-----------------

Address	Publics by Value
---------	------------------

Program entry point at 0000:0100

LINK heeft CODE__SEG vóór DATA__SEG geladen. Dit is precies de volgorde die we willen. Sterker nog: CODE__SEG *moet* het eerste segment in het geheugen zijn, zodat ons programma begint op 100h vanaf het begin van de groep.

Anderzijds kan uw overzicht deze twee segmenten ook in omgekeerde volgorde hebben laten zien. Dat is een teken dat u een Microsoft-versie van de assembler hebt. Als dat het geval is, krijgt u het volgende overzicht te zien:

Warning: no stack segment

Start	Stop	Length	Name	Class
00000H	00000H	00001H	DATA_SEG	
00010H	00111H	00102H	CODE_SEG	

Origin	Group
0000:0	CGROEP

Address	Publics by Name
---------	-----------------

Address	Publics by Value
---------	------------------

Program entry point at 0000:0110

Er klopt niets in dit laadoverzicht. DATA__SEG staat vóór CODE__SEG in het geheugen, en dat betekent dat het statement ORG 100h ons een offset geeft vanaf het einde van DATA__SEG in plaats van vanaf het begin van de groep.

De laatste regel van dit overzicht laat zien dat het beginadres van ons programma nu 110h is. Maar voor een .COM-bestand moet het 100h zijn. Dus wat gebeurt er als we hier een .COM-bestand van proberen te maken?

Draai EXE2BIN en u ziet het volgende:

```
C>EXE2BIN SEGTEST SEGTEST.COM
File cannot be converted
C>
```

Aan die foutmelding hebben we niet veel — ze geeft ons geen enkele aanwijzing waarom we het programma niet kunnen converteren. Maar hier komt het laadoverzicht van pas. Daaruit kunnen we afleiden dat LINK onze segmenten in de verkeerde

volgorde in het geheugen heeft geladen. We hoeven nu alleen nog maar uit te zoeken hoe dit probleem kan worden verholpen. We hebben er in dit boek steeds voor gezorgd dat alle programma's draaien onder zowel de Microsoft- als de IBM-versie van de assembler. Vandaar ook dat we in al onze bronbestanden het datasegment ná het codesegment hebben gezet.

Als u bij uw werk aan assembleerprogramma's programma's maakt of tegenkomt waarin de datasegmenten vooraan in het bestand staan, gebruik dan de parameter `/A` die bij de Microsoft-versie van MASM beschikbaar is. De `/A`-optie vertelt MASM dat u segmenten in alfabetische volgorde geladen wilt hebben. Om deze optie te proberen moet u ons voorbeeldprogramma `SEGTEST.ASM` eens opnieuw assembleren met de volgende opdracht:

```
C>MASM SEGTEST/A;
```

Link dit bestand weer en maak een nieuw laadoverzicht. U moet de twee segmenten nu in alfabetische volgorde zien, met `CODE__SEG` vooraan in het bestand.

C.2 Phase errors

We hebben ervoor gezorgd dat de voorbeelden onder alle versies van de Macro Assembler draaien — van IBM zowel als Microsoft — door het datasegment aan het eind van onze bestanden te zetten. Maar in veel gevallen is dat niet verstandig. In deze paragraaf bekijken we de problemen die eruit kunnen voortvloeien, en geven we betere manieren om uw segmenten te organiseren.

Laten we een concreet voorbeeld nemen:

```
CODE_SEG      SEGMENT PUBLIC
               ASSUME  CS:CODE_SEG, ES:DATA_SEG

BEGIN  PROC    NEAR
        MOV     AX,DATA_SEG           ;haal nummer datasegment op
        MOV     ES,AX                 ;stel ES in op onze gegevens
        MOV     AL,VARIABELE          ;lees 'variabele' in AL
        MOV     AH,4Ch                 ;terug naar DOS
        INT     21h
BEGIN  ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
VARIABELE     DB      0
DATA_SEG      ENDS

               END      BEGIN
```

We hebben het datasegment aan het eind van dit programma gezet om te zorgen dat `DATA__SEG` na `CODE__SEG` in het geheugen wordt geladen. Maar de assembler genereert een *phase error*-melding wanneer we het proberen te assembleren:

```
C>MASM TEST;
```

```
Microsoft (R) Macro Assembler Version 5.00  
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.
```

```
test.ASM(10): error A2006: Phase error between passes
```

```
4486 + 0 Bytes symbol space free
```

```
0 Warning Errors
```

```
1 Severe Errors
```

Wat betekent *phase error*?

Het blijkt dat de Macro Assembler een bestand verschillende keren doorneemt bij het assembleren. Bij de eerste slag (*pass*) verzamelt hij de informatie die hij nodig heeft, zoals het type en segmenten van variabelen. Ter wille van de doelmatigheid begint de assembler het programma bij de eerste slag ook al te assembleren; en daarbij raken wij in de problemen.

MASM assembleert de instructie `MOV AL,VARIABELE` voordat hij weet in welk segment `VARIABELE` staat, dus hij assembleert de `MOV`-instructie alsof we geen segment override (in dit geval `ES`;) nodig hebben. Bij de tweede slag merkt MASM echter dat hij een segment override moet toevoegen omdat `VARIABELE` in het segment staat dat door het `ES`-register wordt aangewezen. Helaas heeft MASM in de eerste slag of *phase* geen ruimte voor deze override-instructie gereserveerd, en geeft hij dus een *phase error*-melding.

We moeten alle variabelen declareren *voordat* we ze in een bestand kunnen gebruiken. Als we dat doen, en we gebruiken de Microsoft Macro Assembler, zal het data-segment vooraan in het geheugen komen te staan, wat bij `.EXE`-bestanden meestal geen probleem is omdat we daarbij meestal met meerdere segmenten werken.

Als u het codesegment echter als eerste in het geheugen geladen wilt hebben, is er een eenvoudige oplossing: zet gewoon een *leeg* segment voor uw datasegment. Het volgende voorbeeld laat zien hoe dat moet.

```
CODE_SEG      SEGMENT PUBLIC          ;laad eerst CODE_SEG  
CODE_SEG      ENDS  
  
DATA_SEG      SEGMENT PUBLIC  
VARIABELE     DB      0  
DATA_SEG      ENDS  
  
CODE_SEG      SEGMENT PUBLIC  
      ASSUME  CS:CODE_SEG, ES:DATA_SEG  
  
BEGIN  PROC   NEAR  
      MOV     AX,DATA_SEG      ;haal nummer datasegment op  
      MOV     ES,AX           ;stel ES in op onze gegevens  
      MOV     AL,VARIABELE     ;lees 'variabele' in AL  
      MOV     AH,4Ch           ;terug naar DOS  
      INT     21h  
BEGIN  ENDP
```



```
CODE_SEG      ENDS

      END      BEGIN
```

C.3 EXE2BIN File cannot be converted

Als u problemen met EXE2BIN hebt, kijk dan eerst in het laadoverzicht of CODE__SEG wel het eerste segment is. Zorg ook dat u maar twee segmenten uitgelijst ziet. Het is wel mogelijk dat er verschillende versies van hetzelfde segment in staan. Bijvoorbeeld:

```
00000H 00103H 00104H CODE_SEG
00110H 00105H 00006H DATA_SEG
00120H 00101H 00102H CODE_SEG
```

In dit geval is CODE__SEG gefragmenteerd. Als u meer dan een deel van een enkel segment in het laadoverzicht ziet, betekent dat dat u problemen hebt die allerlei oorzaken kunnen hebben.

- U hebt misschien geen pseudo-bewerking PUBLIC na al uw segment-definities gezet.
- U hebt misschien iets andere SEGMENT-definities in uw bronbestanden. Kijk al uw bronbestanden na en controleer of alle SEGMENT-definities identiek zijn.
- In een van uw bronbestanden kan een GROUP-statement ontbreken of de GROUP-statement kan foutief zijn. Kijk of alle GROUP-statements wel gelijk zijn.
- Als de GROUP-statements in orde zijn, kijk dan of de ASSUME-statements er zo uitzien:

```
ASSUME CS:CGROEP, DS:CGROEP
```

- U hebt een STACK-segment gedefinieerd. .COM-programma's hebben geen STACK-segment nodig — sterker nog: u *mag* er zelfs geen definiëren.
- Het ingangspunt (*entry point*) is niet 100h. Dat kan komen doordat u de naam van de startprocedure niet na de pseudo-bewerking END in het hoofdbronbestand hebt gezet, of doordat u de bestanden in de verkeerde volgorde hebt gelinkt. De hoofdprocedure *moet* in het eerste bestand staan dat in de LINK-lijst genoemd wordt.

Meer informatie over foutmeldingen en wat ze betekenen vindt u in appendix D.

Appendix D

Veel voorkomende foutmeldingen

D.1 MASM 356

D.2 LINK 357

D.3 EXE2BIN 358

In deze appendix staat een aantal veel voorkomende foutmeldingen die u kunt tegenkomen bij gebruik van MASM, LINK en EXE2BIN. Het is slechts een kleine keuze uit de vele foutmeldingen die deze programma's kunnen genereren. Als u een foutmelding hier niet kunt vinden, kijk dan in de handleiding van uw macro-assembler of DOS. Heeft u een andere versie van MASM, LINK of EXE2BIN, dan kunnen de meldingen enigszins afwijken.

De foutmeldingen zijn in drie groepen verdeeld: een voor MASM, een voor LINK en een voor EXE2BIN. De foutmeldingen zijn in elke paragraaf alfabetisch gerangschikt.

D.1 MASM

Block nesting error

Deze fout komt u waarschijnlijk tegen in combinatie met een melding *Open procedures* of *Open segments*. Zie de beschrijvingen verderop van deze twee meldingen.

End of file, no END directive

Of het END-statement aan het eind van uw bestand ontbreekt, of u moet na het bestaande END-statement nog een blanco regel toevoegen. De Microsoft-versie van de macro-assembler verwacht helemaal aan het eind van het bestand een blanco regel. Als u niet minstens een blanco regel na END hebt, leest MASM het END-statement niet.

Must be declared in pass 1

Deze foutmelding verschijnt meestal in combinatie met een GROUP-statement. Ze betekent dat u een van de segmenten die u in het GROUP-statement hebt gezet, niet hebt gedefinieerd. Als u bijvoorbeeld de regel *CGROEP GROUP CODE__SEG, DATA__SEG* hebt, maar geen datasegment met de naam DATA__SEG hebt gedefinieerd, zult u waarschijnlijk deze melding zien. Ga na of u alle segmenten hebt gedeclareerd die in het GROUP-statement staan.

No or unreachable CS

MASM moet een ASSUME-statement zien om te weten hoe hij sommige instructies, zoals een vertakking of aanroep, moet assembleren. Deze foutmelding betekent dat MASM geen ASSUME-statement heeft kunnen vinden of dat er in het ASSUME-statement dat hij aantrof een fout zat. Kijk in uw bronbestand of er wel een ASSUME-statement in staat, en of die juist is.

Open procedures

Dit betekent dat er of geen PROC- of geen ENDP-statement in het bestand staat, of dat de namen in een PROC/ENDP-paar niet dezelfde zijn. Kijk of PROC een overeenkomstig ENDP-statement heeft, en ga na of de procedurenamen in zowel het PROC- als het ENDP-statement overeenkomen.

Open segments

Er ontbreekt een SEGMENT- of een ENDS-statement, of de namen zijn niet dezelfde in een SEGMENT/ENDS-paar. Zorg dat elk SEGMENT- een overeenkomstig

ENDS-statement heeft, en ga na of de procedurenaam in SEGMENT- en het ENDS-statement overeenkomen.

Symbol not defined

Er zijn drie dingen die u bij deze foutmelding moet nagaan:

1. Misschien hebt u een naam verkeerd getikt. Kijk in de regel die u in de foutmelding ziet of de naam goed is.
2. Misschien hebt u de naam verkeerd getikt toen u een PROC of een variabele de eerste keer declareerde. Vergelijk de namen in de foutieve regel met de namen in de PROC- of variabele-declaraties.
3. Misschien ontbreekt er een EXTRN-declaratie, of is de naam in de EXTRN verkeerd.

D.2 LINK

Fixup offset exceeds field width

Dit is een lastige fout, en vaak het moeilijkst te ontdekken. De melding betekent meestal dat u een procedure als FAR hebt gedeclareerd, maar diezelfde procedure later in een EXTRN-declaratie als NEAR hebt gedeclareerd.

Het kan ook betekenen dat een groep groter is geworden dan de 64K-grens voor groepen. U kunt op beide fouten controleren door het lengteveld in het overzichtbestand na te gaan.

Deze melding kan ook verschijnen wanneer uw segment gefragmenteerd is geraakt. In dergelijke gevallen kunnen de twee fragmenten meer dan 64K van elkaar staan, wat betekent dat aanroepen FAR moeten zijn om te lukken. In appendix C vindt u meer informatie over gefragmenteerde segmenten.

Als dat niet het probleem lijkt, zult u verder moeten speuren. Lees appendix C goed door, maak dan een laadoverzicht van uw programma. Misschien staat daar iets in waar u iets aan hebt. Kijk bijvoorbeeld in welke volgorde de segmenten staan. Misschien staan ze niet in de juiste volgorde.

Symbol defined more than once

Dit betekent dat u dezelfde procedure of variabele waarschijnlijk in twee bronbestanden hebt gedefinieerd. Zorg dat u elke naam in slechts een bronbestand hebt gedefinieerd, gebruik dan EXTRNs op andere plaatsen waar u dezelfde procedure of variabele moet gebruiken.

Unresolved externals

Als u deze melding ziet, ontbreekt er een PUBLIC in het bestand waarin u de procedure of variabele hebt gedeclareerd, of hebt u de naam in een EXTRN-declaratie in een ander bronbestand verkeerd getikt.

Warning: no stack segment

Dit is eigenlijk geen foutmelding, maar gewoon een waarschuwing. U zult deze foutmelding zien bij de voorbeelden in dit boek omdat we .COM-bestanden aanmaken en .COM-bestanden geen afzonderlijk segment voor de stapel gebruiken. Zie hoofdstuk 28 voor een voorbeeldprogramma waarbij LINK deze melding niet geeft.

D.3 EXE2BIN

File cannot be converted

Dit is waarschijnlijk de enige foutmelding die u bij EXE2BIN zult zien, en veel hebt u er niet aan. Meestal betekent het een van deze drie mogelijkheden:

1. Uw segmenten staan in de verkeerde volgorde, zodat er een segment in het geheugen voor `CODE_SEG` staat. Kijk in het laadoverzicht of dit het probleem is. Zie appendix C voor nadere informatie.
2. Uw hoofdprogramma is niet het eerste bestand in uw LINK-lijst. Dat moet wel, dus probeer nog eens opnieuw te linken om zeker te weten dat dit niet het probleem is. Ook hier kunt u dit soort probleem vaak vaststellen door het laadoverzicht te bekijken.
3. In uw hoofdprogramma staat geen `ORG 100h` als eerste statement na de declaratie `CODE_SEG SEGMENT PUBLIC`. Kijk ook of het `END`-statement in uw hoofd-bronbestand het label van de instructie bevat waarmee u wilt beginnen - bijvoorbeeld `END DSKPATCH`.

Als deze suggesties niet helpen, kijk dan appendix C door voor nadere informatie.

Table E-1. ASCII Tekencodes

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0		43	2B	+	86	56	V	129	81	ù
1	1	©	44	2C	,	87	57	W	130	82	ú
2	2	®	45	2D	-	88	58	X	131	83	û
3	3	®	46	2E	.	89	59	Y	132	84	ü
4	4	•	47	2F	/	90	5A	Z	133	85	å
5	5	•	48	30	0	91	5B	[134	86	ä
6	6	•	49	31	1	92	5C	\	135	87	ç
7	7	•	50	32	2	93	5D]	136	88	è
8	8	■	51	33	3	94	5E	^	137	89	ë
9	9	o	52	34	4	95	5F	'	138	8A	è
10	A	■	53	35	5	96	60		139	8B	ï
11	B	ð	54	36	6	97	61	a	140	8C	î
12	C	q	55	37	7	98	62	b	141	8D	ì
13	D	r	56	38	8	99	63	c	142	8E	ñ
14	E	ñ	57	39	9	100	64	d	143	8F	â
15	F	*	58	3A	:	101	65	e	144	90	ê
16	0	»	59	3B	;	102	66	f	145	91	æ
17	11	«	60	3C	<	103	67	g	146	92	æ
18	12	‡	61	3D	=	104	68	h	147	93	ô
19	13	¶	62	3E	>	105	69	i	148	94	ö
20	14	¶	63	3F	?	106	6A	j	149	95	ò
21	15	§	64	40	@	107	6B	k	150	96	û
22	16	_	65	41	A	108	6C	l	151	97	ü
23	17	‡	66	42	B	109	6D	m	152	98	y
24	18	†	67	43	C	110	6E	n	153	99	ÿ
25	19	‡	68	44	D	111	6F	o	154	9A	ü
26	1A	→	69	45	E	112	70	p	155	9B	ç
27	1B	←	70	46	F	113	71	q	156	9C	ç
28	1C	←	71	47	G	114	72	r	157	9D	¥
29	1D	*	72	48	H	115	73	s	158	9E	¥
30	1E	▲	73	49	I	116	74	t	159	9F	f
31	1F	▼	74	4A	J	117	75	u	160	AA	á
32	20		75	4B	K	118	76	v	161	A1	í
33	21	!	76	4C	L	119	77	w	162	A2	ó
34	22	"	77	4D	M	120	78	x	163	A3	ú
35	23	#	78	4E	N	121	79	y	164	A4	ñ
36	24	\$	79	4F	O	122	7A	z	165	A5	ñ
37	25	%	80	50	P	123	7B	{	166	A6	*
38	26	&	81	51	Q	124	7C		167	A7	*
39	27	'	82	52	R	125	7D	}	168	A8	¿
40	28	(83	53	S	126	7E	~	169	A9	¿
41	29)	84	54	T	127	7F		170	AA	¿
42	2A	*	85	55	U	128	80	Ç	171	AB	¿

Tabel E-1. *vervolg*

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
172	AC	¼	193	C1	Ā	214	D6	Ɔ	235	EB	ð
173	AD	½	194	C2	Ĳ	215	D7	Ǝ	236	EC	•
174	AE	¾	195	C3	Ĵ	216	D8	Ǝ	237	ED	•
175	AF	•	196	C4	Ĳ	217	D9	Ǝ	238	EE	€
176	B0	•	197	C5	Ĳ	218	DA	Ǝ	239	EF	•
177	B1	•	198	C6	Ĳ	219	DB	Ǝ	240	F0	≡
178	B2	•	199	C7	Ĳ	220	DC	Ǝ	241	F1	±
179	B3	•	200	C8	Ĳ	221	DD	Ǝ	242	F2	¿
180	B4	•	201	C9	Ĳ	222	DE	Ǝ	243	F3	¿
181	B5	•	202	CA	Ĳ	223	DF	Ǝ	244	F4	Ǝ
182	B6	•	203	CB	Ĳ	224	E0	α	245	F5	Ǝ
183	B7	•	204	CC	Ĳ	225	E1	ρ	246	F6	÷
184	B8	•	205	CD	Ĳ	226	E2	Ǝ	247	F7	•
185	B9	•	206	CE	Ĳ	227	E3	Ǝ	248	F8	•
186	BA	•	207	CF	Ĳ	228	E4	Σ	249	F9	•
187	BB	•	208	D0	Ĳ	229	E5	σ	250	FA	•
188	BC	•	209	D1	Ǝ	230	E6	Ǝ	251	FB	Ǝ
189	BD	•	210	D2	Ǝ	231	E7	Ǝ	252	FC	•
190	BE	•	211	D3	Ǝ	232	E8	Ǝ	253	FD	•
191	BF	•	212	D4	Ǝ	233	E9	θ	254	FE	•
192	C0	•	213	D5	Ǝ	234	EA	Ω	255	FF	•

Veel van de toetsen op het toetsenbord (zoals de functietoetsen) geven een code van twee tekens door wanneer u de toetsen via DOS leest; een decimale 0 gevolgd door een scancode. Hieronder staan de scancodes voor alle toetsen die geen overeenkomstige ASCII-code hebben.

Tabel E-2. Uitgebreide toetsenbordcodes

15	Shift-Tab
16-25	Alt-toetsen voor Q, W, E, R, T, Y, U, I, O, P
30-38	Alt-toetsen voor A, S, D, F, G, H, J, K, L
44-50	Alt-toetsen voor Z, X, C, V, B, N, M
59-68	F1 t/m F10
71	Home
72	Cursor omhoog
73	PgUp
75	Cursor naar links
77	Cursor naar rechts
79	End
80	Cursor omlaag
81	PgDn
82	Ins
83	Del
84-93	Shift-F1 t/m -F10
94-103	Control-F1 t/m -F10
104-113	Alt-F1 t/m -F10
114	Control-PrtSc
115	Control-Cursor naar links
116	Control-Cursor naar rechts
117	Control-End
118	Control-PgDn
119	Control-Home
120-131	Control-Alt voor 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	Control-PgUp

Tabel E-3. De adresseermodi

Adresseermodus	Vorm adres	Gebruikt segmentregister
Register	register (zoals AX)	Geen
Onmiddellijk	gegeven (zoals 12345)	Geen
<i>Geheugenadresseermodi</i>		
Indirect register	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Indirect met basis*	label[BX]	DS
	label[BP]	SS
Direct geïndexeerd*	label[DI]	DS
	label[SI]	DS
Geïndexeerd met basis*	label[BX + SI]	DS
	label[BX + DI]	DS
	label[BP + SI]	SS
	label[BP + DI]	SS
Stringopdrachten (MOVSW, LODSB, enz.)		Lezen uit DS:SI Schrijven naar ES:DI

* Label[...] kan worden vervangen door [verpl + . . .], waarbij *verpl* een verplaatsing is. We zouden dus [10 + BX] kunnen schrijven en het adres zou dan 10 + BX zijn.

Tabel E-4. INT 10h-functies

(AH)=0

Videomodus instellen.

Het AL-register bevat het modusnummer.

Tekstmodi

(AL)=0	40 bij 25, zwart-wit
(AL)=1	40 bij 25, kleur
(AL)=2	80 bij 25, zwart-wit
(AL)=3	80 bij 25, kleur
(AL)=7	80 bij 25, monochrome video-adapter

Grafische modus

(AL)=4	320 bij 200, kleur
(AL)=5	320 bij 200, zwart-wit
(AL)=6	640 bij 200, zwart-wit

(AH)=1

Cursorgrootte instellen.

(CH)	Eerste beeldlijn van de cursor. De bovenste lijn is 0 op zowel de monochrome als kleur/grafische schermen, terwijl de onderste lijn 7 is voor de kleur/grafische adapter en 13 voor de monochrome adapter. Geldig bereik: 0 t/m 31.
(CL)	Laatste beeldlijn van de cursor.

De instelling bij aanzetten bij de kleur/grafische adapter is CH = 6 en CL = 7. Voor het monochrome scherm: CH = 11 en CL = 12.

(AH)=2

Cursorpositie instellen.

(DH,DL)	Rij, kolom van nieuwe cursorpositie; de linker bovenhoek is (0,0).
(BH)	Paginanummer. Dit is het nummer van de schermpagina. De kleur/grafische adapter heeft plaats voor meerdere schermpagina's, maar de meeste programma's gebruiken pagina 0.

(AH)=3

Cursorpositie lezen.

(BH)	Paginanummer
retour: (DH,DL)	Rij, kolom van cursor
(CH,CL)	Cursorgrootte

Tabel E4. *vervolg*

(AH)=4	Positie lichtpen lezen (zie <i>Technical Reference-handleiding</i>).
(AH)=5	Actieve schermpagina selecteren.
(AL)	Nieuwe paginanummer (van 0 t/m 7 voor modi 0 en 1; van 0 t/m 3 voor modi 2 en 3)
(AH)=6	Pagina omhoog schuiven.
(AL)	Aantal leeg te maken regels onderaan op het scherm. Bij normaal verschuiven wordt er een regel gewist. Stel op 0 in om hele venster leeg te maken
(CH,CL)	Rij, kolom van linker bovenhoek van venster
(DH,DL)	Rij, kolom van rechter benedenhoek van venster
(BH)	Schermattribuut voor blanco regels
(AH)=7	Pagina omlaag schuiven.
	Zelfde als pagina omhoog schuiven (functie 6), maar regels blijven bovenaan op het scherm leeg in plaats van onderaan.
(AH)=8	Lees attribuut en teken onder de cursor.
(BH)	Schermpagina (alleen tekstmodi)
(AL)	Gelezen teken
(AH)	Attribuut van gelezen teken (alleen tekstmodi)
(AH)=9	Schrijf attribuut en teken onder de cursor.
(BH)	Schermpagina (alleen tekstmodi)
(CX)	Aantal keren dat teken en attribuut naar scherm moeten worden geschreven
(AL)	Te schrijven teken
(BL)	Te schrijven attribuut
(AH)=10	Schrijf teken onder cursor (bij normaal attribuut).
(BH)	Schermpagina
(CX)	Aantal keren dat teken moet worden geschreven
(AL)	Te schrijven teken
(AH)=11 t/m 13	Verschillende grafische functies. (Zie <i>Technical Reference-handleiding</i> voor nadere bijzonderheden.)

Tabel E-4. vervolg

(AH)=14

Schrijf telex.

Schrijf een teken naar het scherm en zet de cursor naar de volgende plaats.

- | | |
|------|--|
| (AL) | Te schrijven teken |
| (BL) | Kleur van teken (alleen grafische modus) |
| (BH) | Schermpagina (tekstmodus) |

(AH)=15

Huidige videostatus doorgeven.

- | | |
|------|-------------------------|
| (AL) | Toon huidige modus |
| (AH) | Aantal tekens per regel |
| (BH) | Actieve scherpagina's |

De nu volgende tabel bevat de INT 21h-functies die in dit boek worden gebruikt. Een uitgebreidere opsomming vindt u in de *DOS Technical Reference*-handleiding. Al deze functies komen tevens uitgebreid aan de orde in het boek *MS-DOS voor gevorderden* van Ray Duncan, uitgave Kluwer Technische Boeken, 1987, ISBN 90 201 1992 3.

Tabel E-5. INT 21h-functies

(AH)=1	<p>Toetsenbord-invoer. Deze functie wacht tot u een teken op het toetsenbord hebt getikt. Ze stuurt het teken naar het scherm en zet de ASCII-code in het AL-register. Bij uitgebreide toetsenbordcodes geeft deze functie twee tekens door: een ASCII 0 gevolgd door de scancode (zie Afb. E-2).</p>
(AL)	Van het toetsenbord gelezen teken.
(AH)=2	<p>Uitvoer afbeelden. Toont een teken op het scherm. Verschillende tekens hebben een bijzondere betekenis voor deze functie.</p>
7	Pieptoon: stuur een toon van een seconde naar de speaker.
8	Backspace: zet de cursor een tekenpositie naar links.
9	Tab: ga naar de volgende tab-positie. Tab-stops worden ingesteld op elk achtste teken.
0Ah	Line feed: ga naar volgende regel.
0Dh	Carriage return: ga naar begin huidige regel.
(DL)	Op scherm af te beelden teken.
(AH)=8	<p>Toetsenbord-invoer zonder echo. Leest een teken van het toetsenbord, maar beeldt het teken niet af op het scherm.</p>
(AL)	Van het toetsenbord gelezen teken.
(AH)=9	<p>String afbeelden. Beeldt de string af die wordt aangewezen door het register-paar DS:DX. U moet het einde van de string aangeven met het teken \$.</p>
DS:DX	Wijst naar de af te beelden string.

Tabel E-5. *vervolg*

(AH)=0AH	Lees string. Leest een string van het toetsenbord. Zie hoofdstuk 23 voor nadere bijzonderheden.
(AH)=4Ch	Terug naar DOS. Keert terug naar DOS, net als INT 20h, maar werkt bij zowel .COM- als .EXE-programma's. De functie INT 20h die in dit boek wordt gebruikt, werkt alleen bij .COM-programma's.
(AL)	Terugkeer-code. Is normaliter 0, maar u kunt hem instellen op elk ander getal en de DOS-batch-opdracht IF en ERRORLEVEL gebruiken om fouten vast te stellen.

Index

A

ADC-instructie 59
ADD-instructie 37
aftrekken 39
algemene registers 34
AND 72
A(ssemble) (DEBUG) 50
Assembler-instructies
 ADC 59
 ADD 37
 CMP 68
 INC 81
 JA 85
 JB 85
 JC 249
 JL 70
 JNE 115
 JNZ 115
 JNZ 67
 JZ 67
 LEA 167
 POP 83
 PUSH 83
 RCL 58
 RET 81
 SHL 77
 SHR 71
 SUB 40
 XOR 114
ASSUME 103, 127

B

binaire getallen 18
bit 29

breakpoint 63
byte 30

C

CF (Carry Flag) 58
CGROEP 215
CLD 181
CMP-instructie 68
code segment (CS) 38
commentaar 97
CURSOR.ASM 191
CX-register 60

D

DB (define byte) 123
DEBUG 18
DEBUG-opdrachten
 A(ssemble) 50
 D(ump) 54
 G(o) 52
 P(roceed) 63
 U(nassemble) 50
 W(rite) 53
decimaal omzetten in hex 25
Define Byte (DB) 123
delen 41
Destination Index (DI) 114
DI (destination index) 114
directe geheugenadressering 152
DISK_IO.ASM 165
DIV 44
Dskpatch 135
DSKPATCH.ASM 194
D(ump) (DEBUG) 54
DWORD 215

E

EDIT__BYTE 236
END 94
ENDP 103
EXEMOD 291
EXTRN 167

F

FANTOOM__OMHOOG 278
FANTOOM__OMLAAG 278
fantoomcursor 204, 222
FAR 103, 128
Foutmeldingen 356

G

GELEZEN__AANTAL__TEK 245
grafische tekens 172
G(o) (DEBUG) 52

H

hex omzetten in decimaal 21
hex-cijfer afdrukken 68
hexadecimale getallen 18
hexarithmetic 19
hoge byte 40

I

INC-instructie 81
indirecte adresseermodus 151
instruction pointer (IP) 38
INT 20h 49
INT 21h 47
INT 25h 167
INT 26h 168
interrupt 47

J

JA-instructie 85
JB-instructie 85
JC-instructie 249
JL-instructie 70
JNE-instructie 115
JNZ-instructie 67, 115
JZ-instructie 67

L

laadoverzicht 268
labels 36
lage byte 40
LEA-instructie 167
LEES__BYTE 245
LEES__SECTOR 165, 192
LEES__STRING 255
LIFO-structuur 82
load map 268
LODSB 181
logische instructies 72

M

Make 162
meervoudige segmenten 284
MODEL 94
MODEL huge 94
MODEL large 94
MODEL medium 94
MODEL small 94
modulair ontwerpen 108, 143
MUL 41

N

NEAR 103, 128
nibble 71
nulvlag 66

O

offset 36
omzetten, decimaal naar hex 25
omzetten, hex naar decimaal 21
optellen 37
ORG 100h 125
ORG 103
overflow 28
overloop 28
overloopvlag 66

P

Phase error between passes 296
POP-instructie 83
PROC 103
P(roceed) (DEBUG) 63
Program Segment Prefix (PSP) 125
pseudo-ops 94
PSP (Program Segment Prefix) 125
PUBLIC 107
public 269
PUSH-instructie 83

R

RCL-instructie 58
registers 34
rekenen met hex-getallen 189
relocatie 287
responsbestand 268
RET-instructie 81
ROM-BIOS 188

S

scan-code 76
schermgeheugen 294
SCHRIJF_ATTRIBUTUT_N_KEER 231
SCHRIJF_FANTOOM 227
SCHRIJF_PROMPTREGEL 212
SCHRSTER.COM 52
screen swapping 272
segment override 294
segmenten 35

segmentregisters 122
SHL-instructie 77
SHR-instructie 71
SI (source index 114
Source Index (SI) 114
speciale registers 34
stack 122
Stack Pointer (SP) 81
Stack Segment (SS) 81 stapel 122
START_SEC.AFB 182, 226
STRING_NAAR_HOOFDLET 245
statusbits 66
STUUR_CRLF 155
SUB-instructie 40
Symdeb 163, 268

T

TBD_IO.ASM 245
Tekenaattributen veranderen 231
tekenvlag 67
TOON_HALVE_SECTOR 155, 164
TOON_SEC.ASM 179, 181
twee-complement 31

U

U(nassemble) (DEBUG) 50

V

VERDEEL_TABEL 214
VERDELER 212, 237, 262
vermenigvuldigen 41
VIDEO_IO.ASM 105
VOLGENDE_SECTOR 217
VORIGE_SECTOR 217

W

woord 30
W(rite) (DEBUG) 53

X

XOR-instructie 114

De tijd dat het maken van programma's voor de PC uitsluitend was voorbehouden aan de professionele programmeur ligt achter ons. Ook de serieuze hobbyist begeeft zich steeds verder in het duistere innerlijk van zijn PC.

Hoewel er inmiddels vele krachtige hogere programmeertalen beschikbaar zijn voor de IBM-PC en compatibelen (BASIC, Pascal, C enz.) is het programmeren in machinetaal voor veel mensen een grote uitdaging. Van belang is daarbij ook de, in vergelijking met hogere programmeertalen, ongeëvenaarde uitvoeringssnelheid van machinetaalroutines.

Peter Norton, de 'goeroe' van de IBM-PC, laat zich in deze uitgave bijstaan door John Socha, een erkende autoriteit op het gebied van 8086-machinetaal.

In dit boek, dat zich kenmerkt door een heldere, op zelfstudie gerichte, schrijfstijl, wordt een stap-voor-stap-uitleg gegeven van begrippen als assemblers, registers, binair rekenen en wat dies meer zij. Er wordt verder veel aandacht geschonken aan het gestructureerd ontwerpen van programma's en aan het gebruik van programmeerhulpmiddelen.

Als een rode draad door het boek heen loopt de ontwikkeling van een volledige toepassing, waarbij alle aspecten van het programmeren in machinetaal aan bod komen.

De programmalistings zijn tevens op diskette verkrijgbaar.